N°20 | SEPTEMBRE - OCTOBRE 2017





HACKABLE MAGAZINE

DÉMONTEZ | COMPRENEZ | ADAPTEZ | PARTAGEZ

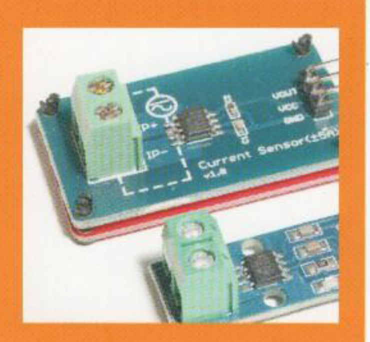
France MÉTRO. : 7,90 € – CH : 13 CHF – BEL/LUX/PORT.CONT : 8,90 € – DOM/TOM : 8,50 € – CAN : 14 $ CAD

DOMOTIQUE / COURANT

Mesurez le courant électrique domestique avec une carte Arduino et un capteur CT ou à effet Hall

p. 84





PLATINE / ÉCONOMIES

Faites des économies en fabriquant simplement vos propres câbles/jumpers pour vos platines à essais

p. 04



RASPBERRY PI / LEDS

Visualisez en direct la charge processeur de votre Pi avec le Pimoroni Piglow et Python

p. 38





Arduino / ESP8266 / DIY

Créez une veilleuse qui montre les phases de la lune

p. 12

- Comprenez les limitations des cartes Arduino
- Adaptez du code venant d'une autre plateforme
- Contrôlez les leds et simulez les phases
- Construisez votre lune lumineuse

RASPBERRY PI / HAT

Affichez des graphismes avancés, du texte et des animations avec la matrice de leds du Sense Hat

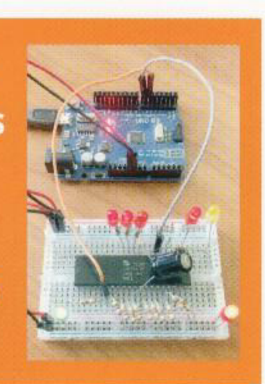
p. 50



VINTAGE / CPU

Faites vos premiers pas dans la construction d'un ordinateur 8 bits Z80 contrôlé par Arduino

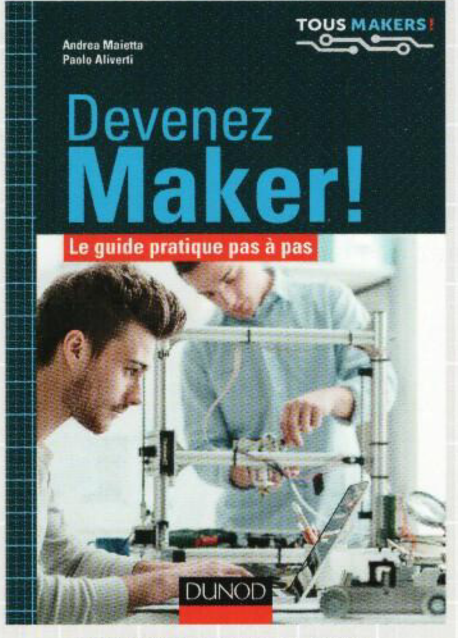
p. 70



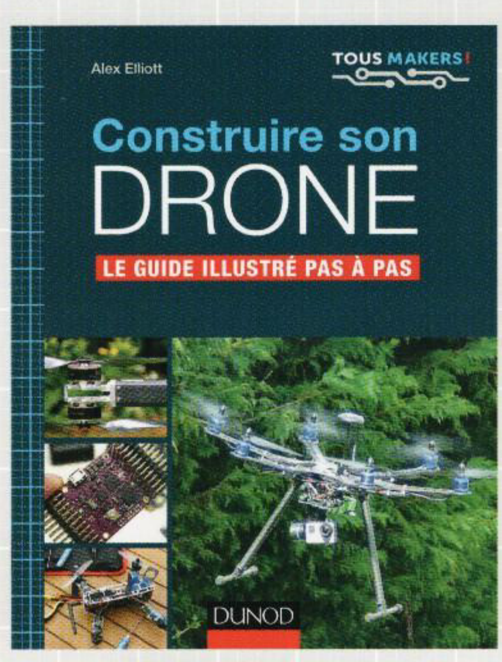


TOUS MAKERS !

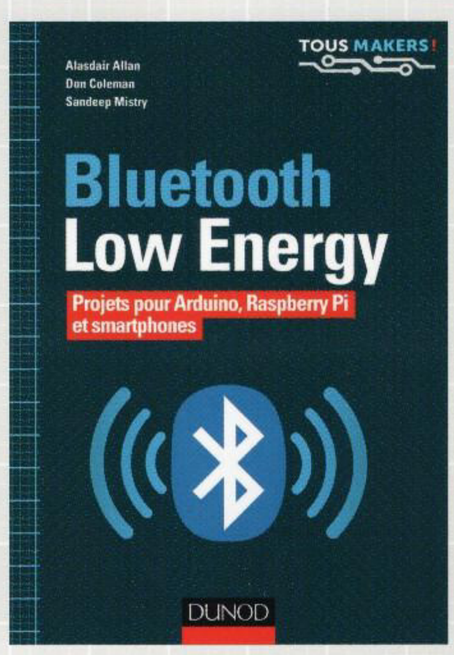
RÉVÉLEZ VOTRE POTENTIEL PAR LA CRÉATION



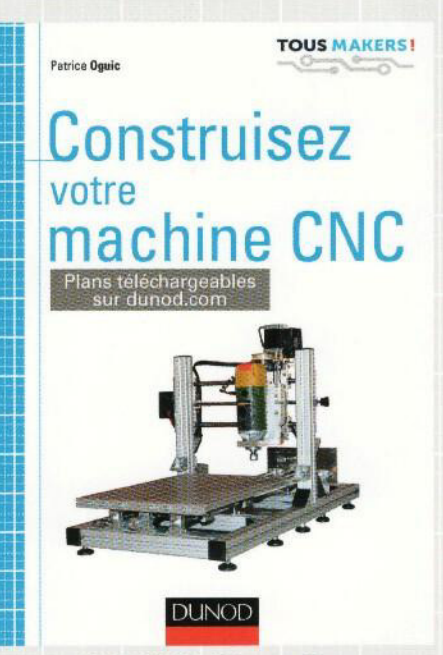
9782100762934, 304 pages, 24,90 €
ANDREA MAIETTA, PAOLO ALIVERTI
Comment transformer vos idées en projets concrets ? Ce livre vous accompagne dans la réalisation de vos premières créations.



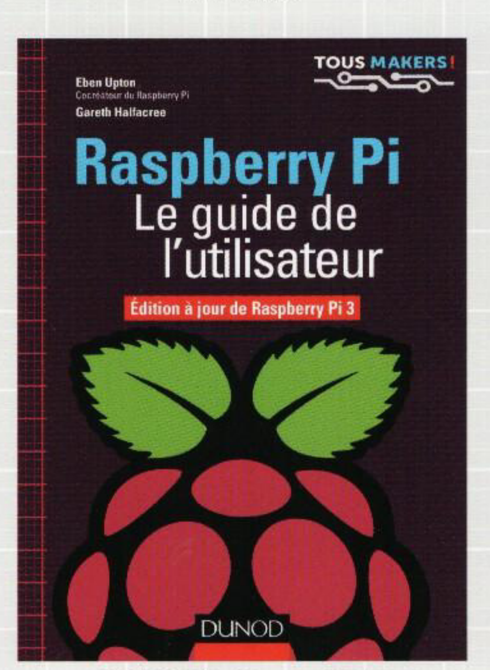
9782100758487, 240 pages, 27 €
ALEX ELLIOTT
Le compagnon idéal pour vous guider pas à pas tout au long de la construction de votre drone.



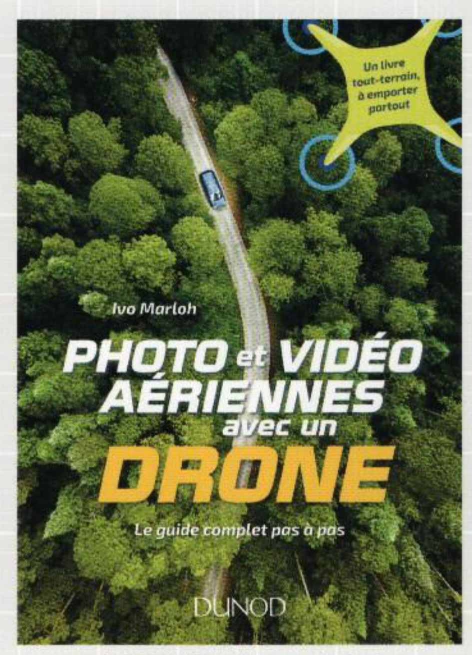
9782100760855, 272 pages, 29 €
ALASDAIR ALLAN ET AL.
Maîtrisez la nouvelle technologie Bluetooth Low Energy en réalisant les différents projets détaillés dans cet ouvrage.



9782100738106, 176 pages, 23 €
PATRICE OGUIC
Un guide pour construire votre machine CNC, idéale pour la gravure et le perçage des circuits imprimés, les maquettistes et les modélistes.



9782100762262, 288 pages, 26,90 €
EBEN UPTON, GARETH HALFACREE
Écrit par le co-créateur du Raspberry Pi, cet ouvrage donne toutes les clés pour tirer le meilleur parti du nano-ordinateur révolutionnaire.



9782100757992, 160 pages, 19,90 €
IVO MARLOH
Toutes les clés pour filmer et photographier avec son drone en toute sérénité.

TOUT LE CATALOGUE SUR **WWW.DUNOD.COM**



ÉDITO



Tout change et rien ne change...

Dans ce numéro, vous trouverez un article un peu particulier traitant d'un processeur dont la conception remonte à plus de 40 ans. Toutefois, celui-ci fonctionne dans les grandes lignes exactement sur les mêmes bases qu'un modèle dernier cri comme l'AMD Ryzen Threadripper (qui, soit dit en passant, semble être une merveille. C'est plaisant de voir AMD revenir dans la course).

Plus d'intégration, plus de vitesse, plus de cœurs, plus de fonctionnalités, plus de bits... mais au final, les mêmes fondations et la même logique. Ceci n'a rien d'exceptionnel et, si l'on regarde de près (de très très près), on se rend rapidement compte qu'entre un SoC ARM d'une Raspberry Pi, un Intel Core i9, le microcontrôleur de votre Arduino ou encore le processeur MIPS d'un routeur ou d'une box, au bout du compte, il n'y a pas vraiment de différences, seul l'usage change. Celui que vous, vous décidez d'en faire.

Python sur Raspberry Pi, C/C++ sur Arduino, assembleur sur un Z80... En réalité, la seule chose qui change, c'est votre comportement, vos préférences. Ceci est tout aussi vrai dans bien des domaines qui nous occupent ici, la radio logicielle, la bidouille, l'électronique, la programmation, le bricolage... mais plus largement encore, c'est vrai dans toutes les disciplines. Et si vous allez à contre-courant, à l'opposé des usages et des habitudes, et obtenez le résultat escompté ou plus encore, vous arrivez dans le hacking, au sens noble du terme : une solution innovante, créative, originale et montrant une certaine excellence technique (n'en déplaise aux médias qui nous martèlent le pénible hacker=pirate).

Et grâce à cela, à cette continuité, nous pouvons améliorer notre compréhension du présent grâce à des choses qui peuvent aujourd'hui paraître « primitives ». Et si cela est vrai pour la technologie, cette « chose » qui fait de nous des humains, ça l'est très certainement aussi pour tout le reste. Je crois que si on expliquait l'importance qu'a l'Histoire ainsi aux plus jeunes, le monde serait bien différent. Mais cela est un tout autre débat...

Sur cette note hautement « philosophique », je vous laisse découvrir ce numéro de rentrée. Les vacances sont finies, au boulot ! ;)

Denis Bodor

Hackable Magazine
est édité par Les Éditions Diamond



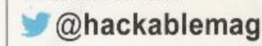
10, Place de la Cathédrale - 68000 Colmar
Tél. : 03 67 10 00 20 – Fax : 03 67 10 00 21
E-mail : lecteurs@hackable.fr
Service commercial : cial@ed-diamond.com
Sites : https://www.hackable.fr/
https://www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Réalisation graphique : Kathrin Scali
Responsable publicité : Valérie Fréchard,
Tél. : 03 67 10 00 27 v.frechard@ed-diamond.com
Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Landau, Allemagne
Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes : Abomarque : 09 53 15 21 77
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution,
N° ISSN : 2427-4631
Commission paritaire : K92470
Périodicité : bimestriel
Prix de vente : 7,90 €



La rédaction n'est pas responsable des textes, illustrations et photos qui lui sont communiqués par leurs auteurs. La reproduction totale ou partielle des articles publiés dans Hackable Magazine est interdite sans accord écrit de la société Les Éditions Diamond. Sauf accord particulier, les manuscrits, photos et dessins adressés à Hackable Magazine, publiés ou non, ne sont ni rendus, ni renvoyés. Les indications de prix et d'adresses figurant dans les pages rédactionnelles sont données à titre d'information, sans aucun but publicitaire. Toutes les marques citées dans ce numéro sont déposées par leur propriétaire respectif. Tous les logos représentés dans le magazine sont la propriété de leur ayant droit respectif.

Suivez-nous sur Twitter



À PROPOS DE HACKABLE...

HACKS, HACKERS & HACKABLE

Ce magazine ne traite pas de piratage. Un **hack** est une solution rapide et bricolée pour régler un problème, tantôt élégante, tantôt brouillonne, mais systématiquement créative. Les personnes utilisant ce type de techniques sont appelées **hackers**, quel que soit le domaine technologique. C'est un abus de langage médiatisé que de confondre « pirate informatique » et « hacker ». Le nom de ce magazine a été choisi pour refléter cette notion de **bidouillage créatif** sur la base d'un terme utilisé dans sa définition légitime, véritable et historique.

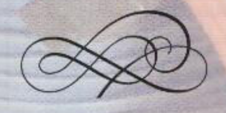
SOMMAIRE





FABRIQUEZ VOS CÂBLES/JUMPERS POUR PLATINES À ESSAIS

Denis Bodor



L'électronique de loisir est un hobby souvent relativement peu coûteux, en particulier lorsqu'on achète des composants, des modules et des accessoires sur des sites de discount ou d'enchères en ligne. Il n'en reste pas moins qu'il n'y a pas de petites économies et que tantôt, un peu d'huile de coude ne fait pas de mal. Si, en plus, cela permet d'avoir un gros stock pour les années à venir et que le résultat est bien plus pratique, on gagne sur tous les tableaux...

Souvent je me dis que les idées simples ou les petites astuces ne méritent pas un article, en particulier lorsque cela concerne une pratique qui m'est devenu quotidienne. C'est en discutant alors avec d'autres personnes, en particulier mes chers lecteurs croisés lors de salons, de manifestations ou de portes ouvertes que je me rends finalement compte que cela pourrait être moins désuet que je ne l'imagine.

Le cas qui nous intéresse aujourd'hui est celui des câbles, connecteurs, *jumper-wire*... utilisés généralement avec les platines à essais. Lorsqu'on débute, on acquiert généralement une ou deux platines et une sympathique boîte contenant des sortes de « fils de fer » gainés permettant d'établir les liaisons proprement. Invariablement, on finit par en manquer, les tordre dans tous les sens, voire les utiliser pour autre chose, quitte à les souder ou s'en servir comme antenne, bobine ou électroaimant.

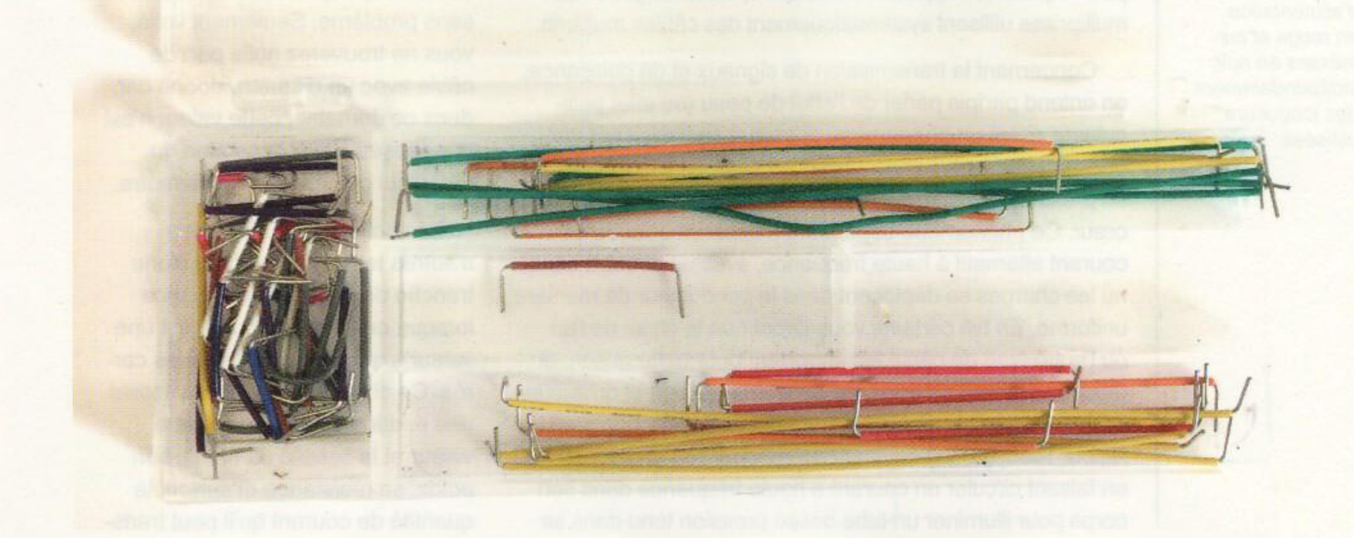
Se pose alors la question de l'achat d'une nouvelle boîte avec, à la clé, invariablement, le même parcours et donc le même problème. Même à moins de 3€ la boîte de 140 pièces de tailles différentes en provenance de Chine, à force, cela finit par coûter le prix d'une carte Arduino. Une autre solution pourtant assure un stock pour une longue durée, un choix de couleurs plus étoffé et surtout une économie substantielle sur le long terme : réaliser ses câbles soi-même.

1. TYPES ET TAILLES DE CÂBLES : C'EST PAS SI SIMPLE !

L'idée de base n'est pas compliquée, il suffit d'acheter plusieurs mètres de câble, de les couper à la taille souhaitée et d'en dénuder les extrémités. Mais on se heurte alors à un petit problème qui prend la forme d'une simple question : quel type de câble choisir ?

La première chose à prendre en considération est la structure du câble en question. On parle généralement de câbles électriques souples (comme une rallonge électrique) ou rigides (comme les câbles dans les murs), mais il est plus juste d'utiliser les termes « conducteurs monobrin » et « conducteurs multibrin ». Le câble monobrin se compose d'un corps d'une pièce couvert d'un isolant.

Voilà typiquement le genre de câble, souvent appelé « jumper wire » qui est utilisé sur les platines à essais. Répartis en plusieurs tailles en quantité limitée, ceux-ci posent toujours le problème typique des « lots » : il n'y a jamais assez de ce dont on a besoin et trop de ce qui ne sert pas souvent.





11 rouleaux de 20m de câble monobrin gainé pour un coût d'environ 40 euros. Non seulement ceci est bien plus rentable que l'achat de boites complètes mais, en plus, la couleur n'est pas liée à la taille. On peut donc, par exemple, avoir toutes les lignes d'alimentation en rouge et les masses en noir, indépendamment des longueurs utilisées.

Le câble multibrin contient plusieurs conducteurs de petits diamètres, torsadés ou non, surmoulés avec un isolant. Le matériau utilisé comme conducteur peut être du cuivre, de l'aluminium, du fer ou un alliage composé de ces métaux et d'autres (or, argent, nickel, magnésium, silicium, etc.). Il existe une grande variété de conducteurs, mais le plus souvent il s'agit simplement de cuivre étamé.

Le choix entre multibrin et monobrin dans le domaine électrique ou hifi est généralement une question de pratique et/ou tantôt de théories plus ou moins fumeuses. Les câbles monobrin ne doivent pas subir de courbures ou de torsions répétées, car cela fragilise le conducteur qui finit invariablement, à un moment ou un autre, par casser. C'est la raison pour laquelle les appareils électriques, les rallonges et les multiprises utilisent systématiquement des câbles multibrin.

Concernant la transmission de signaux et de puissance, on entend parfois parler de l'effet de peau (ou effet pelliculaire). C'est un phénomène physique se traduisant par une circulation plus importante des charges électriques en proximité avec surface du conducteur et non dans leur cœur. Ce phénomène cependant n'apparaît qu'avec un courant alternatif à haute fréquence, avec un courant continu les charges se déplacent dans le conducteur de manière uniforme. En hifi certains vous diront que le choix de l'un ou l'autre type de câble est important pour cette raison, et d'autres affirmeront que c'est de la superstition et qu'aucune différence audible n'existe. Pour l'anecdote, sachez que Nikola Tesla utilisait ce phénomène pour ses présentations, en faisant circuler un courant à haute fréquence dans son corps pour illuminer un tube basse pression tenu dans sa main. Du fait de l'effet de peau, le courant provenant d'une bobine ne pénétrait pas dans son corps et il ne risquait donc pas l'électrocution, mais le résultat avait le mérite d'étonner son auditoire... Un peu comme les discours et arguments sans fin sur les forums audiophiles, concernant le type de câbles à utiliser pour tel ou tel style musical.

Dans le cas qui nous concerne ici, l'effet de peau ou la souplesse des câbles n'a presque aucune importance. En effet, quiconque a déjà essayé d'enfoncer un câble multibrin dans une platine à essais connaît parfaitement le problème. C'est un peu comme essayer de passer un fil de laine dans le chas d'une aiguille sans le mouiller... en pire. Pour fabriquer vos câbles pour platines à essais, il vous faut donc du câble monobrin (ou *solid core* en anglais). La question mérite à peine d'être posée.

Reste ensuite à déterminer le diamètre du câble en question afin que celui-ci entre bien dans le trou, mais ne soit pour autant pas trop mince et qu'il puisse tenir en place sans problème. Seulement voilà, vous ne trouverez nulle part de câble avec un diamètre donné car, dans ce domaine, cette valeur n'est pas utilisée. C'est la section du câble qui compte, non le diamètre.

La section du câble, ou en d'autres termes, la surface d'une tranche de la partie conductrice lorsque celui-ci est coupé, est une valeur exprimée en millimètres carrés. Ce choix a été fait, car il existe une relation directe entre cette valeur et la solidité du câble, son poids, sa résistance et surtout la quantité de courant qu'il peut trans-

porter. Différentes sections normalisées (IEC 60228) existent de 0,5 mm², 0,75 mm², 1 mm²... jusqu'à 2500 mm². Un certain nombre de normes imposent, pour les installations électriques, des sections d'une valeur spécifique en fonction de la distance et de la puissance de courant à transporter.

Convertir un diamètre en section n'est pas très difficile, il suffit de se souvenir de ses cours de géométrie de primaire : surface d'un cercle = rayon fois rayon fois pi. Et donc, section = (diamètre / 2)² x pi = pi x diamètre² / 4. Inversement, on pourra obtenir le diamètre à partir de la surface en calculant la racine carrée de la surface fois deux divisée par pi.

Cette désignation en mm² est cependant spécifique aux pays utilisant le système métrique (dont nous). Les USA et implicitement donc un grand nombre de vendeurs sur le net préfèrent un autre système : AWG pour *American Wire Gauge* (littéralement « calibre de fils américain »).

Alors que la conversion d'un diamètre en section, et inversement, relève d'un calcul relativement simple, il n'en va pas de même pour l'AWG. En premier lieu, contrairement à la relation diamètre/section, la valeur AWG est proportionnellement inverse à la mesure du diamètre. Plus la valeur AWG est grande, plus le diamètre du conducteur est petit.

Initialement le créateur de ce système, la société Brown & Sharpe, a arbitrairement défini deux valeurs, basées sur deux diamètres : 0000 AWG (ou 4/0 AWG) pour le diamètre de 0,46 pouces et 36 AWG pour 0,005 pouces. De cette base a été calculée une liste de 40 valeurs suivant une progression géométrique (comme la section est multipliée par une constante pour obtenir chaque valeur, le diamètre lui, augmente de façon dite géométrique). Cette gamme a, par la suite, été étendue pour aller jusqu'à 40 AWG.



Il est possible de faire les calculs pour déduire le diamètre d'une valeur AWG et inversement, mais ceci est relativement fastidieux. Il existe d'ailleurs des astuces pour avoir une approximation, comme par exemple le fait que le diamètre d'un conducteur est doublé lorsque la valeur AWG est réduite par 6 ou encore que réduire de 3 la valeur AWG double la section d'un câble. Mais le plus simple est de tout bonnement se référer à un tableau présentant les correspondances AWG, diamètre et section comme celui ci-contre :

AWG	Diamètre en pouce	Diamètre en mm	Section en mm²
0000 (4/0)	0,46	11,684	107
000 (3/0)	0,4096	10,405	85,0
00 (2/0)	0,3648	9,266	67,4
0 (1/0)	0,3249	8,251	53,5
1	0,2893	7,348	42,4
2	0,2576	6,544	33,6
3	0,2294	5,827	26,7
4	0,2043	5,189	21,2
5	0,1819	4,621	16,8
6	0,162	4,115	13,3
7	0,1443	3,665	10,5
8	0,1285	3,264	8,37
9	0,1144	2,906	6,63
10	0,1019	2,588	5,26
11	0,0907	2,305	4,17
12	0,0808	2,053	3,31
13	0,072	1,828	2,62
14	0,0641	1,628	2,08
15	0,0571	1,45	1,65
16	0,0508	1,291	1,31
17	0,0453	1,15	1,04
18	0,0403	1,024	0,823
19	0,0359	0,912	0,653
20	0,032	0,812	0,518
21	0,0285	0,723	0,41
22	0,0253	0,644	0,326
23	0,0226	0,573	0,258
24	0,0201	0,511	0,205
25	0,0179	0,455	0,162
26	0,0159	0,405	0,129
27	0,0142	0,361	0,102
28	0,0126	0,321	0,081
29	0,0113	0,286	0,0642
30	0,01	0,255	0,0509
31	0,00893	0,227	0,0404
32	0,00795	0,202	0,032
33	0,00708	0,18	0,0254
34	0,0063	0,16	0,0201
35	0,00561	0,143	0,016
36	0,005	0,127	0,0127
37	0,00445	0,113	0,01
38	0,00397	0,101	0,00797
39	0,00353	0,0897	0,00632
40	0,00314	0,0799	0,00501

Dans le cas des câbles rigides pour platines à essais, il s'agit généralement de conducteurs monobrin en 22 AWG et donc avec un diamètre de 0,644 mm et une section de 0,326 mm². Vous remarquerez également qu'un câble 21 AWG ou 23 AWG ne présente pas de grande différence et pourrait tout aussi bien convenir. Mais si nous restons dans les valeurs initiales, tout ce que nous avons à faire est de rechercher du câble gainé de ce type, ce qui se traduit généralement en une utilisation des termes « *solid wire 22 awg breadboard jumpers* ».



Cette pince coupante spécialement conçue pour l'électronique dispose, à l'arrière, d'un ingénieux mécanisme pour dénuder les câbles avec deux petites mâchoires qui pincent le câble et deux autres, coupantes, qui se rétractent pour retirer la gaine isolante. Ce modèle d'Amtech coûte environ 15€, mais pour un câble monobrin un simple cutter pourra parfaitement faire l'affaire.

Une telle recherche sur eBay, par exemple, devrait vous retourner un certain nombre de résultats intéressants à la fois de vendeurs en Europe et en Asie. En ce qui me concerne, j'ai fait mon achat chez un vendeur anglais (*mallinson-electrical*). Je dis bien « mon achat », au singulier, car celui-ci portait sur un lot de 11 rouleaux de couleurs différentes de 20 mètres, le tout pour un montant d'une quarantaine d'euros. Avec plus de 200m de câble, autant vous dire que ceci risque de me durer un petit bout de temps...

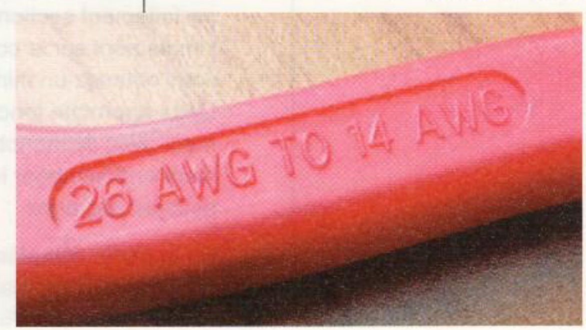


2. UTILISER LES CÂBLES

Réceptionner un lot de câbles bruts et s'extasier devant les belles couleurs offertes est une chose, en faire des « jumpers » pouvant être utilisés sur platine à essais en est une autre. Pour transformer la matière brute en quelque chose d'utilisable il faut : couper, dénuder et plier.

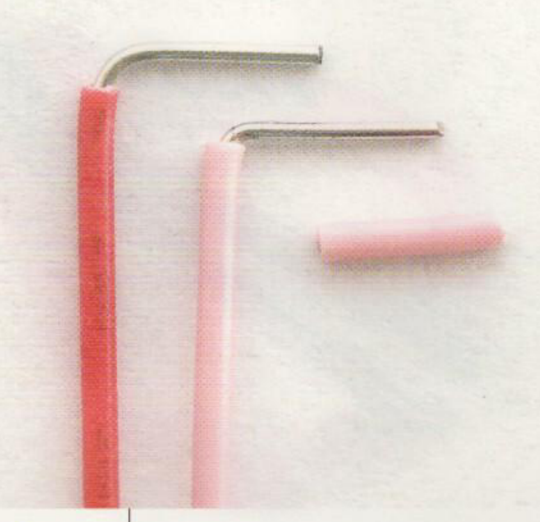
Personnellement, je ne prépare pas de « jumpers » à l'avance, mais préfère consommer le câble au fur et à mesure. Le coupage du câble et éventuellement la pliure de ce dernier ne posent pas vraiment de problème. Un outillage de qualité sera un plus mais, si nécessaire, les moyens du bord feront l'affaire. J'ai d'ailleurs toujours un coupe-ongles à portée de main sur mon bureau, ceci fonctionne très bien et tantôt même mieux qu'une pince hors de prix.

La plupart des mesures de câbles qu'on trouve sur le net lorsqu'on cherche du matériel ou des fournitures à acheter utilisent le système AWG. Ici, la poignée d'une pince coupante précise des calibres utilisables entre 26 et 14 AWG, soit une section entre 0,129 mm² et 2,08 mm² et donc un diamètre entre 0,405 mm et 1,63 mm.



Dénuder le câble est une autre paire de manches et plusieurs techniques existent :

- Utiliser une pince à dénuder. Généralement, ceci fonctionne très bien avec des câbles multibrin, mais n'est souvent pas optimal avec du monobrin.
- Utiliser une pince coupante standard en ajustant judicieusement la fermeture des mâchoires. Cette technique est relativement difficile à maîtriser et demande pas mal de pratique. Il faut arriver à contrôler sa main



Voici le résultat visé lorsque vous décidez de réaliser vos « jumpers » avec à gauche (en rouge) un exemplaire provenant d'une boîte et à droite (en rose) la version « faite maison » avec le morceau de gaine retiré.

de façon à pincer le câble juste ce qu'il faut pour entamer l'isolant et appliquer un mouvement de traction pour le retirer... Le tout sans couper le conducteur ou étirer simplement l'isolant sans l'enlever. Cette technique, à mon sens, est plus adaptée aux câbles souples multibrin.

- Utiliser un briquet, ce qui généralement est une vieille technique d'électricien, donnant des résultats assez douteux et surtout étant relativement douloureuse. L'idée est de chauffer le bout du câble pour ramollir la gaine et la retirer d'un coup sec en la prenant entre le pouce et l'index. Ne le faites pas, c'est barbare et fort désagréable.
- Et enfin, ma méthode préférée qui consiste à utiliser un objet coupant comme un cutter ou un couteau. Il suffit de rouler le câble sur la lame avec une pression suffisante pour couper la gaine, mais sans endommager le conducteur (qui est généralement assez tendre puisqu'il s'agit en principe de cuivre). Le morceau d'isolant est ensuite retiré, parfaitement sectionné, en le glissant tout simplement sur le conducteur (et en plus vous obtenez un mini macaroni tout mignon). Cette approche fonctionne merveilleusement avec du monobrin, mais peut facilement endommager les câbles souples en coupant des brins.

Plier la partie dénudée du câble n'est, bien entendu, pas difficile, mais surtout, elle n'est pas forcément nécessaire. Rien ne vous oblige en effet à créer des connexions « à plat » sur la platine même si pour certaines d'entre elles cela peut être plus pratique (connexion à la masse ou à la ligne d'alimentation, sur le bord de la platine par exemple).

3. POUR FINIR

Les câbles monobrin de ce genre de calibre peuvent être utilisés pour bien des choses et non pas simplement comme « jumpers » sur platine. Vous pouvez par exemple créer des bobines, des antennes, des électroaimants, etc. Ils présentent généralement une limitation, du moins en 22 AWG, dans le sens où ils peuvent difficilement être utilisés pour une connexion directe sur une carte Arduino par exemple.

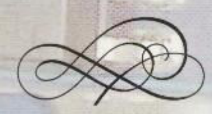
En effet, les emplacements sur la carte sont trop grands (ou le diamètre du conducteur trop petit, c'est selon) et même en choisissant un câble avec un diamètre plus important, leur rigidité ne facilite pas la connexion avec une platine à essais et/ou un module. Dans ce genre de situations, l'option du câble souple avec des connecteurs adaptés (souvent appelés câbles « Dupont ») est préférable. Il est possible de tricher en donnant une forme en dent de scie au conducteur monobrin, mais le résultat est loin d'être idéal.

Comme dans d'autres domaines, finalement, pour chaque usage il existe une solution adaptée et optimale. À vous de la choisir en fonction de ce que vous avez sous la main et du but à atteindre. **DB**

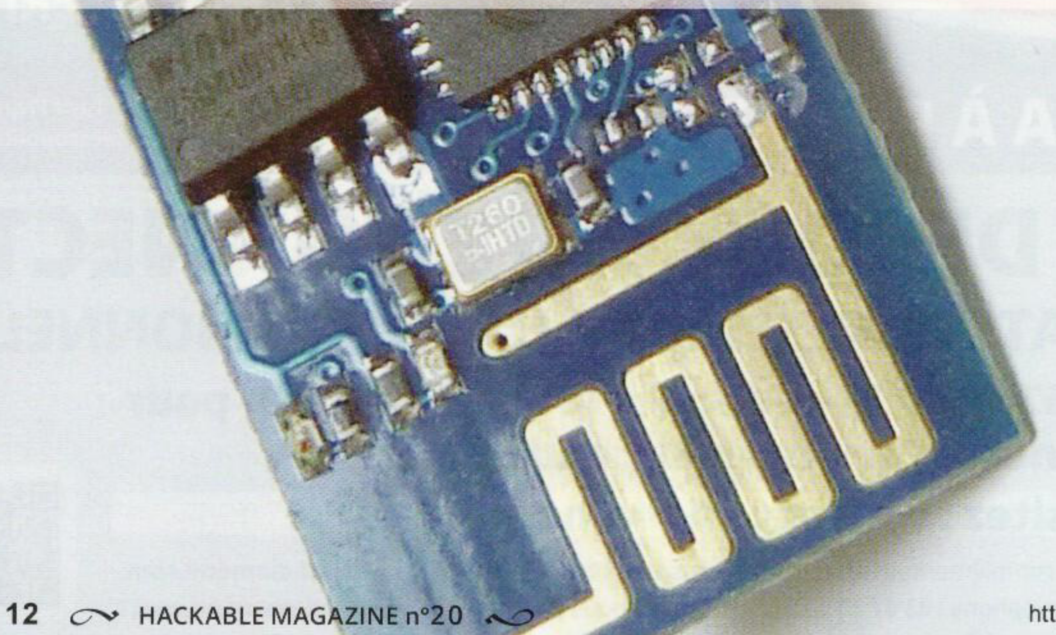


CRÉEZ UNE LAMPE LUNAIRE : PRÉPARATION

Denis Bodor



On peut s'intéresser à la lune pour de nombreuses raisons, de la plus raisonnable à la plus farfelue, mais quelle qu'elle soit il faut avouer que ce satellite naturel de la Terre est du plus bel effet dans le ciel. Pourquoi donc alors le laisser à sa place et ne pas le rapporter chez vous, dans votre chambre ou dans votre salon ? Les budgets spatiaux n'étant pas dans les plages utilisables pour le magazine, nous allons nous contenter de reproduire l'astre avec la plus grande précision possible.



Une fois n'est pas coutume, ce projet est également une aventure et l'occasion d'apprendre énormément. Ce qui a commencé par une simple idée amusante, « tiens ce serait drôle d'avoir une lune lumineuse qui suit les phases en temps réel », a presque failli finir en véritable cauchemar. En effet, et c'est sans doute pour cela qu'on dit que la lune est trompeuse, en regardant un calendrier tout cela paraît fort simple : il suffit de suivre le cycle lunaire et incrémenter petit à petit au fil du temps qui passe... Eh bien non, pas du tout !

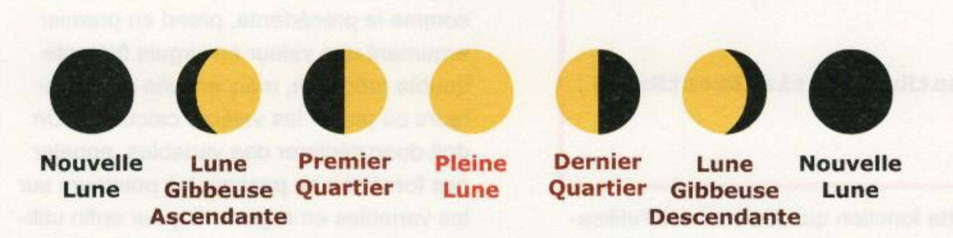
Figurez-vous que la lune ne va pas toujours à la même vitesse, qu'elle n'est pas toujours à la même distance de la Terre, que la Terre a une position tout aussi variable par rapport au soleil... et qu'en mélangeant tout cela à grands coups de calculs trigonométriques, c'est presque la migraine assurée, du moins si on n'est pas amateur d'astronomie (ce que je ne suis aucunement, je trouvais simplement l'idée rigolote).

Avec le recul, il eut été plus simple de calculer l'illumination lunaire à partir d'un site ou d'une application pour les 15 prochaines années, et d'enregistrer tout cela dans une carte Arduino accompagnée d'un petit croquis de lecture. Mais non, voyez-vous, je me suis mis en tête de chercher un bout de code en C que je pourrais réutiliser, porter sur Arduino et étendre à souhait par la suite. En effet, l'idée même de devoir assimiler toute une littérature grouillant de termes étranges et de transformer tout cela en code me paraissait un peu... démesurée (jetez simplement un œil à la page Wikipédia de l'équation de Kepler et vous comprendrez).

1. LE CODE DE DÉPART

J'ai ainsi mis la main sur le code d'un certain John Walker datant de 1987, repris par Bill Randle en 1991 afin de l'intégrer dans un outil appelé `calentool` et faisant partie de FreeBSD 1.0 (1993). En domaine public, ce code en C, prenant la forme d'un fichier `moon.c`, était un composant d'un outil utilisant XView/Xlib, une « interface » graphique (*toolkit*) à la mode à l'époque. La première phase du travail consista donc principalement à débarrasser le code de toute la gestion graphique puis de le modifier pour obtenir une première version utilisable en ligne de commandes sous GNU/Linux. Les résultats calculés pouvaient être ensuite comparés avec des données disponibles sur différents sites, ainsi que celles fournies par des applications Android par exemple.

Pour adapter un code existant de cette manière, la technique est relativement simple, mais fastidieuse. On commence tout naturellement par supprimer les instructions incluant les fichiers d'entête `.h` inutiles (ce qu'on appelle généralement des bibliothèques dans l'environnement Arduino). On tente ensuite de compiler le code et on utilise les erreurs générées pour retrouver les fonctions posant problème et devant être supprimées. Tout ceci pourrait être fait dans l'environnement Arduino, mais je trouve que le cycle compilation/correction est plus rapide sur un PC GNU/Linux ou une Raspberry Pi.



Voici les différentes phases de la lune au cours d'une lunaison. Petite astuce pour savoir si la lune est croissante ou décroissante : « la lune est menteuse », lorsque la partie lumineuse forme un « C » la lune est Décroissante, lorsqu'elle forme un « D » elle est Croissante...

Une fois toutes les erreurs éliminées, on active ensuite les messages d'avertissement du compilateur (option `-Wall`) de façon à afficher toutes les variables déclarées et non utilisées. Là encore le processus est relativement simple, mais néanmoins pénible. Il s'agit principalement de se débarrasser de la majeure partie du « code mort » de manière à obtenir une version la plus édulcorée possible, constituée du strict minimum utile.

Enfin, il faut étudier le code restant, car un certain nombre de déclarations inutiles ne sont pas détectées par le compilateur. En effet, si je déclare une variable `toto` en début d'une fonction par exemple et que, plus loin, je lui attribue une valeur avec un `toto=12`, le compilateur ne la considèrera bien entendu pas comme non utilisée, même si elle ne sert à rien ailleurs dans le code. Là, la seule recommandation pour ce travail consiste à ne pas vous précipiter et recompiler à chaque variable traitée afin de vous assurer que vous n'avez effectivement rien raté. Si vous supprimez un lot de variables et obtenez une erreur concernant l'une d'entre elles, il sera problématique de revenir en arrière sans annuler tous les changements.

Dans ce cas précis, pour le fichier `moon.c`, le code n'était qu'un élément d'un programme plus important. Il n'avait pas de fonction principale (`main()`) permettant l'exécution à proprement parler. Dans le cas d'un croquis Arduino, cette fonction vous est masquée, mais elle existe également. Elle est constituée de quelque chose comme ceci :

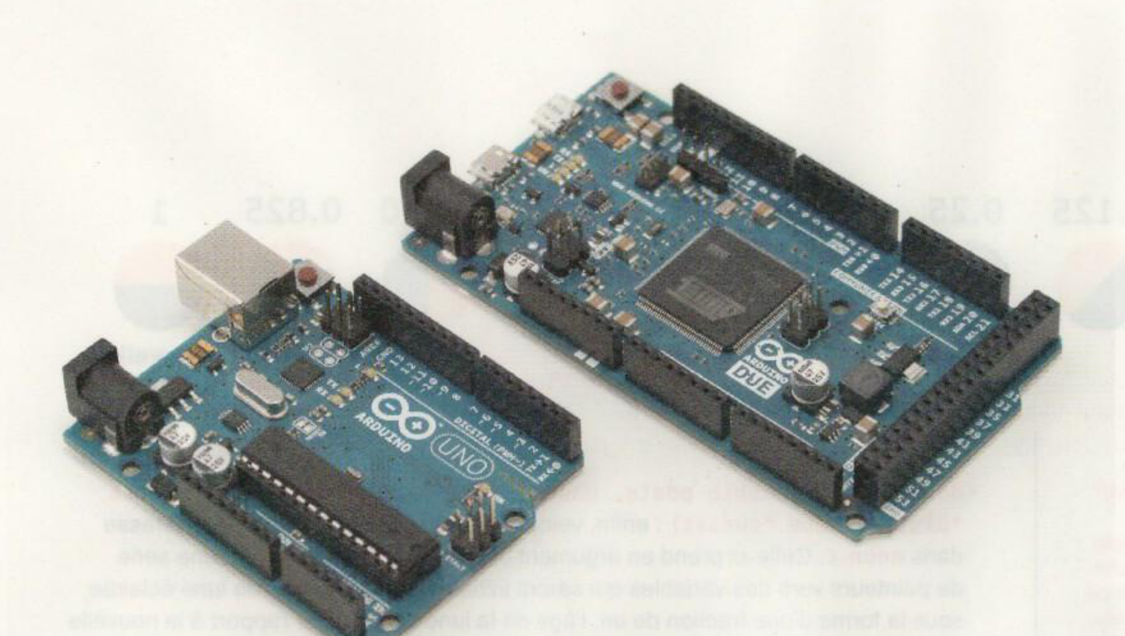
```
int main(void)
{
  setup();

  for (;;) {
    loop();
    if (serialEventRun) serialEventRun();
  }
  return 0;
}
```

C'est cette fonction qui déclenchera l'utilisation de celles que nous voulons tester. Comme cette partie de l'article se veut la plus générique possible, je n'entrerai pas dans le détail de la façon d'écrire un code en C fonctionnant, par exemple, sur Raspberry Pi. Vous pourrez trouver sur le dépôt GitHub du magazine le fichier source `moon.c` modifié et complété ainsi d'un fichier `Makefile` permettant de simplifier la compilation (placez-vous simplement dans le répertoire et utilisez la commande `make` pour obtenir un binaire `moon`).

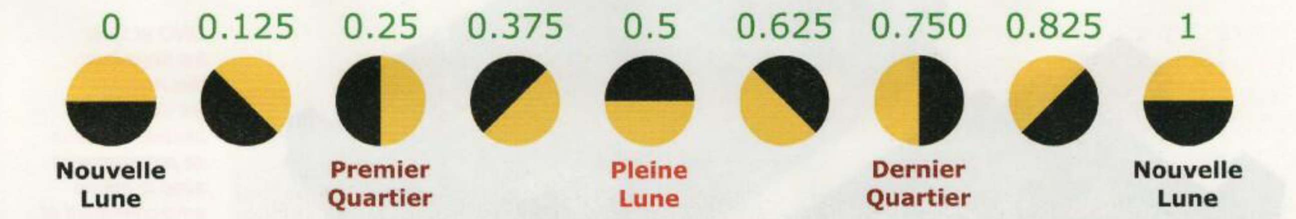
Au final, ce qui reste du code original se résume aux fonctions suivantes (notez que les noms des fonctions et des variables sont ceux choisis par l'auteur original du code) :

- `void jyear(double td, int *yy, int *mm, int *dd)` : permet de convertir une date exprimée en jours juliens en trois entiers représentant l'année, le mois et le jour. Le jour julien est la base d'un système de datation utilisé en astronomie et qui consiste à représenter une date en un nombre de jours et une fraction de jours écoulés depuis une date fixée par convention. Normalement cette date est le 1er janvier -4712 à 12 heures, mais le CNES utilise le 1er janvier 1950 à 0 heure et la NASA le 24 mai 1968 à 0 heure, date de début des missions Apollo (les Américains ne peuvent pas s'empêcher de fanfaronner apparemment). Le code de John Walker, lui, utilise la date conventionnelle, le *jour julien astronomique* ou AJD en anglais.
- `void jhms(double j, int *h, int *m, int *s)` : fait la même chose que `jyear()`, mais permet d'obtenir l'heure, les minutes et les secondes d'une date en jours juliens. Notez que cette fonction, comme la précédente, prend en premier argument une valeur en virgule flottante double précision, mais ensuite des pointeurs où placer les valeurs calculées. On doit donc déclarer des variables, appeler ces fonctions en passant les pointeurs sur les variables en argument, pour enfin utiliser ces variables ainsi initialisées.



UNO et Due : les deux sont des Arduino, les deux sont bleues, les deux se programment avec le même environnement et le même langage... mais ces deux cartes sont vraiment très différentes. Celle de droite utilise des « double » et une arithmétique en 64 bits et l'autre en 32 bits seulement.

- **`double meanphase(double sdate, double k)`** : retourne la date « approximative » de la nouvelle lune en jours juliens pour une date donnée en argument ainsi que le numéro du mois lunaire (ou synodique) depuis l'an 1900. Notez qu'un mois lunaire synodique, également appelé une lunaison, est l'espace entre deux nouvelles lunes (totalement obscur), soit environ 29,53 jours (oui, utiliser une date de nouvelle lune et calculer des intervalles de 29,53 est une solution simple pour déduire les phases de la lune, mais cela reste approximatif).
- **`double truephase(double k, double phase)`** : permet d'obtenir une date et heure exacte d'une phase exprimée en fraction de un (0 étant la nouvelle lune, 0,25 le premier quartier, 0,5 la pleine lune, etc.). **`k`** est ici, comme avec la fonction précédente, le numéro de mois lunaire depuis 1900.
- **`void phasehunt(double sdate, double phases[5])`** : permet d'obtenir cinq phases lunaires (nouvelle lune, premier quartier, pleine lune, dernier quartier, nouvelle lune) entourant une date donnée en jour julien. Cette fonction utilise à la fois **`meanphase()`** pour trouver les mois lunaires adéquats puis **`truephase()`** pour calculer les dates et heures précises des cinq phases. Notez que, comme avec **`jyear()`** et **`jhms()`**, le tableau permettant de stocker ces cinq dates doit être déclaré avant l'appel à la fonction et c'est un pointeur sur cette variable qui est passé en argument (**`phase`** et **`phase[0]`** pointant sur la même donnée).
- **`double kepler(double m, double ecc)`** : calcule l'équation de Kepler permettant de déterminer la position des astres pour un instant donné sur leurs orbites elliptiques. Cette fonction est utilisée par la fonction principale du fichier **`moon.c`** (**`phase()`**).
- **`long jdate(struct tm *t)`** : retourne la partie entière du jour julien pour une date donnée sous la forme d'une structure standard **`tm`** définie dans **`time.h`** sur un système Unix comme GNU/Linux. Cette structure est sensiblement différente dans l'environnement Arduino et cette fonction doit être adaptée.
- **`double jtime(struct tm *t)`** : retourne la partie entière et décimale du jour julien pour une date donnée sous la forme d'une structure **`tm`**. Cette fonction utilise la précédente et retourne donc une valeur en virgule flottante.



La valeur retournée par la fonction phase() est suffisante pour notre projet puisqu'elle fournit à elle seule une excellente indication de l'illumination de la lune pour une date donnée. Ici, il faut s'imaginer être au-dessus de la lune, avec un observateur se trouvant en bas de l'illustration. La rotation indiquée par phase() nous donne à la fois la portion illuminée, mais également notre position temporelle dans le cycle lunaire (lune croissante ou décroissante).

- **double phase(double pdate, double *pphase, double *mage, double *dist, double *sudist)** : enfin, voici la fonction principale qui nous intéresse dans **moon.c**. Celle-ci prend en argument une date en jours juliens et une série de pointeurs vers des variables qui seront initialisées : la quantité de lune éclairée sous la forme d'une fraction de un, l'âge de la lune en jour (par rapport à la nouvelle lune), la distance entre le centre de la Terre et la lune et la distance du soleil en kilomètres. Cette fonction retourne également une valeur correspondante à un angle de rotation entre 0 et 1 représentant un cycle lunaire complet en terme d'illumination. Il faut imaginer cette valeur comme une rotation imaginaire du soleil autour de la lune ou encore comme une balle bicolore (obscure/lumineuse) tournant sur elle-même, et où la rotation est représentée par une valeur entre 0 et 1 (cf. illustration).

Voici pour les fonctions utiles récupérées de **moon.c** qui peuvent directement être utilisées pour obtenir des informations intéressantes en fonction de la date et heure actuelle. Pour l'occasion, j'ai également ajouté une nouvelle fonction, **nextmoon(double sdate, double phase)** directement inspirée de **phasehunt()** et permettant d'obtenir la date d'une prochaine phase en passant en argument la date actuelle et la phase en question sous forme d'une fraction de un :

```
double nextmoon(double sdate, double phase)
{
    int yy, mm, dd;
    double k, ret;

    // calcul de la date en jours juliens
    jyear(sdate, &yy, &mm, &dd);
    // calcul du mois lunaire
    k = floor((yy + ((mm - 1) * (1.0 / 12.0)) - 1900) * 12.3685);

    // on cherche la date de la prochaine phase demandée
    while((ret=truephase(k, phase)) < sdate) {
      k++;
    }

    // on retourne la date trouvée
    return ret;
}
```

On pourra ainsi rapidement obtenir la date et l'heure de, par exemple, la prochaine pleine lune (**0.5**) ou nouvelle lune (**0.0**), mais également de n'importe quelle phase intermédiaire « officielle », voire celle d'une fraction totalement arbitraire. Une fois le code totalement nettoyé et adapté, le résultat obtenu en ligne de commandes sur une machine GNU/Linux comme une Raspberry Pi ressemble à ceci :

```
Maintenant (GMT): Mon Mar  6 10:29:37 2017
Jour julien: 2457818.94
    Lune croissante
    angle = 0.28      [....****]
    illumination = 61%    5/8
    age lunaire = 8.40j
    distance = 365221.12 km
    distance soleil = 148447908.33 km
phase 0: Nouvelle lune = 26/02/2017 15:00:15
phase 1: Premier quartier = 05/03/2017 11:33:38
phase 2: Pleine lune = 12/03/2017 14:54:49
phase 3: Dernier quartier = 20/03/2017 16:01:29
phase 4: Nouvelle lune = 28/03/2017 02:59:26

Prochaine pleine lune = 12/03/2017 14:54:49
Prochaine nouvelle lune = 28/03/2017 02:59:26
```

Toutes ces informations sont bien plus étoffées que celles pouvant être utiles pour un montage simple éclairant une lune en plastique ou un disque imprimé, mais nous avons une base solide avec des résultats confirmés par différents sites, logiciels et applications Android. Il ne reste plus qu'à transférer tout ça dans l'environnement Arduino.

2. DES « DOUBLE » PAS VRAIMENT « DOUBLE »

En adaptant rapidement le code pour en faire un croquis Arduino, on se rend compte d'un gros problème. Cette adaptation consiste principalement à remplacer les appels à la fonction `printf` par `Serial.print()`, obtenir la date et l'heure actuelle via une horloge temps réel type DS1307/DS1338 et, bien entendu, à réorganiser le code de la fonction `main()` pour le placer dans `setup()`. Ce n'est pas un travail très difficile puisque la majorité des fonctions originales sont directement utilisables sans changement. C'est l'affaire de quelques minutes, mais...

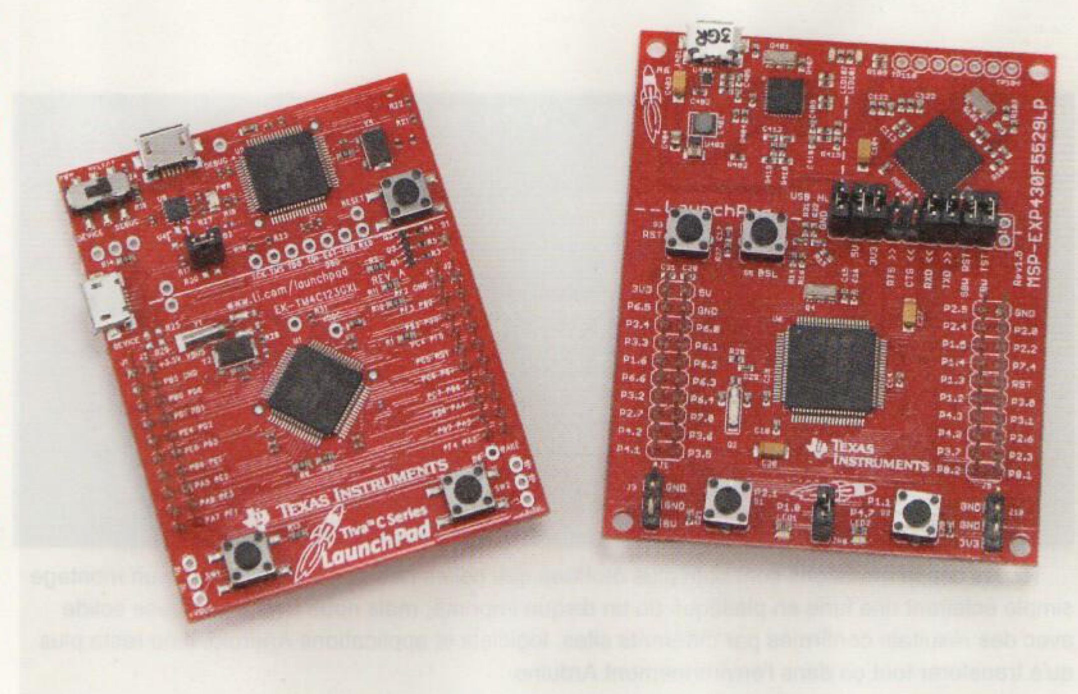
Lors de l'exécution sur une carte Arduino UNO, par exemple, les informations affichées laissent dubitatif :

```
Maintenant (GMT): 06/03/2017 10:30:11
Jour julien: 2457818.25
  Lune Croissante
  angle: 0.26
  illumination: 53%    5/10
  age lunaire: 7.67j
  distance: 364056.78km
  distance soleil: 148421536.00km
Nouvelle lune = 26/02/2017 12:00:00
Premier quartier = 05/03/2017 12:00:00
Pleine lune = 12/03/2017 12:00:00
Dernier quartier = 20/03/2017 18:00:00
Nouvelle lune = 28/03/2017 00:00:00

Prochaine pleine lune = 12/03/2017 12:00:00
Prochaine nouvelle lune = 28/03/2017 00:00:00
```

Toutes les valeurs sont différentes, et ce dans une mesure qui va bien au-delà de l'écart de temps entre l'exécution sur une Pi et celle sur Arduino. Pire encore, les cinq valeurs retournées par

Le problème de précision sur les valeurs à virgule flottante (ou « les flottants ») n'est pas spécifique à Arduino. Le MSP430 et son compilateur intégré à Energia se comportent de la même manière, les « double » sont en réalité des « float ». À droite, une Launchpad MSP-EXP430F5529LP avec des « double » en 32 bits et à gauche une Launchpad Tiva C TM4C123G avec des vrais « double ».



phasehunt(), une fois converties au format du calendrier civil, affichent des heures étrangement « rondes ».

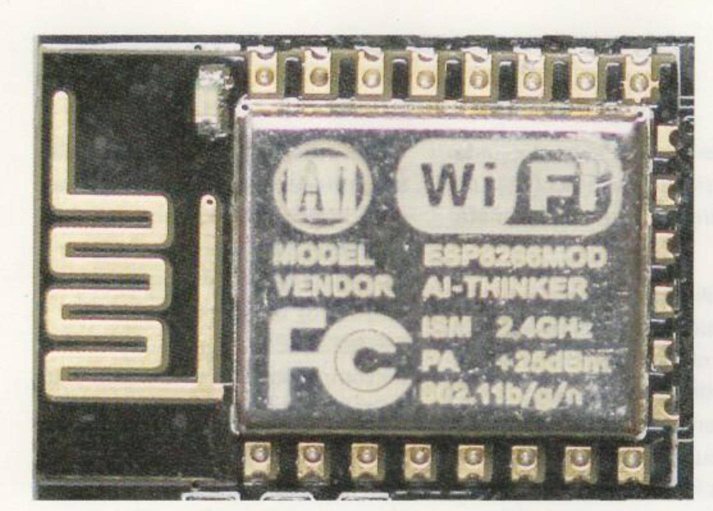
Ce sont pourtant exactement les mêmes fonctions qui sont utilisées et le problème ne vient certainement pas du code utilisé pour l'affichage. Il s'agit donc clairement d'une différence concernant la plateforme elle-même ou son support logiciel.

En réalité, nous venons de toucher une limitation importante de la plateforme et plus exactement des cartes Arduino reposant sur un microcontrôleur AVR. Celui-ci intègre en effet un processeur 8 bits possédant des limitations importantes en termes de calcul. L'une d'elles concerne la taille en bits des types de variables destinées à stocker des valeurs en virgule flottante. Cette limitation est intimement liée à la fois au compilateur et aux bibliothèques standards (libc) pour cette architecture, sans être strictement liée au processeur lui-même, si ce n'est en termes de performances.

Lorsque je parle de précision, ceci concerne un type bien particulier utilisé tout du long dans le code que nous avons récupéré : le type **double** utilisé pour stocker des valeurs réelles (à virgule flottante) en double précision. Ce type est normalement censé être normalisé (IEEE 754) selon une représentation sur 64 bits (1 bit de signe, 8 bits d'exposant et 52 bits de mantisse) et on parle alors d'une précision de 53 bits. Or dans le cas du compilateur œuvrant en coulisse de l'environnement Arduino, les **double** sont en réalité des **float** (simple précision), codés sur 32 bits (1 bit de signe, 8 bits d'exposant et 23 bits de mantisse).

Je n'entrerai pas ici dans le détail de la manière dont un ordinateur, au sens large du terme, peut stocker une valeur en virgule flottante, mais retenez simplement que plus le nombre de bits est important, plus la précision l'est tout autant. Le nom des différents types utilisé est une chose, et le format exact en est une autre. Les notions de simple précision et double précision sont trop vagues pour clarifier la situation et une norme a été établie. Il existe donc maintenant des désignations précises : *binary32*, *binary64* et *binary128*. Le fait est que, avec une carte et un compilateur pour AVR, que ce soit avec des types **float** ou **double**, c'est du *binary32* qui est utilisé.

Essayez simplement ce croquis et vous en aurez le cœur net :



Voici un ESP-12E monté sur une carte NodeMCU. Sous le blindage en métal se cache un microcontrôleur ESP8266 à base de Tensilica Xtensa LX106, un processeur 32 bits à 80 Mhz. Cette plateforme économique peut être utilisée simplement avec l'environnement Arduino et propose des vrais « double » binary64 codés sur 64 bits, avec une précision de 53 bits. Exactement ce qu'il nous faut pour nos calculs lunaires. Et en prime, l'ESP8266 dispose d'une connectivité Wifi !

```
void setup() {
  Serial.begin(115200);

  Serial.print("char : ");
  Serial.println(sizeof(char)*8);
  Serial.print("int : ");
  Serial.println(sizeof(int)*8);
  Serial.print("long : ");
  Serial.println(sizeof(long)*8);
  Serial.print("float : ");
  Serial.println(sizeof(float)*8);
  Serial.print("double : ");
  Serial.println(sizeof(double)*8);
}

void loop() {
}
```

Dans le moniteur série s'affichera, lors de l'exécution :

```
char : 8
int : 16
long : 32
float : 32
double : 32
```

Faites de même avec une carte Arduino Due par exemple, et :

```
char : 8
int : 32
long : 32
float : 32
double : 64
```

Nous n'avons rien changé au croquis, c'est l'environnement de développement (compilateur, etc.) et les bibliothèques utilisés qui changent.

Le code initial provenant de BSD utilise des **double** pour une bonne raison, mais l'environnement Arduino pour Atmel AVR ne nous propose que des « faux » **double** (il en va de même pour les cartes Launchpad MSP430 de Ti et l'environnement Energia). Dans l'état, tout ce travail d'adaptation aura-t-il donc été fait pour rien ?

3. LE PROBLÈME ET LA SOLUTION

Les calculs opérés par le code sont relativement complexes et je n'ai pas honte de l'avouer, me sont totalement incompréhensibles à ce stade. L'objectif même de la démarche consistant à trouver et adapter un code existant était justement de ne pas avoir à me documenter et prendre le temps d'assimiler des informations et principes pour réimplémenter tout l'algorithme.

Trois solutions sont alors envisageables pour régler ce problème :

- Réécrire une grande partie du code pour essayer, au mieux, de corriger le problème de précision en basant tous les calculs sur des **float** de 32 bits. La tâche n'est pas impossible, mais relativement fastidieuse et surtout impose de

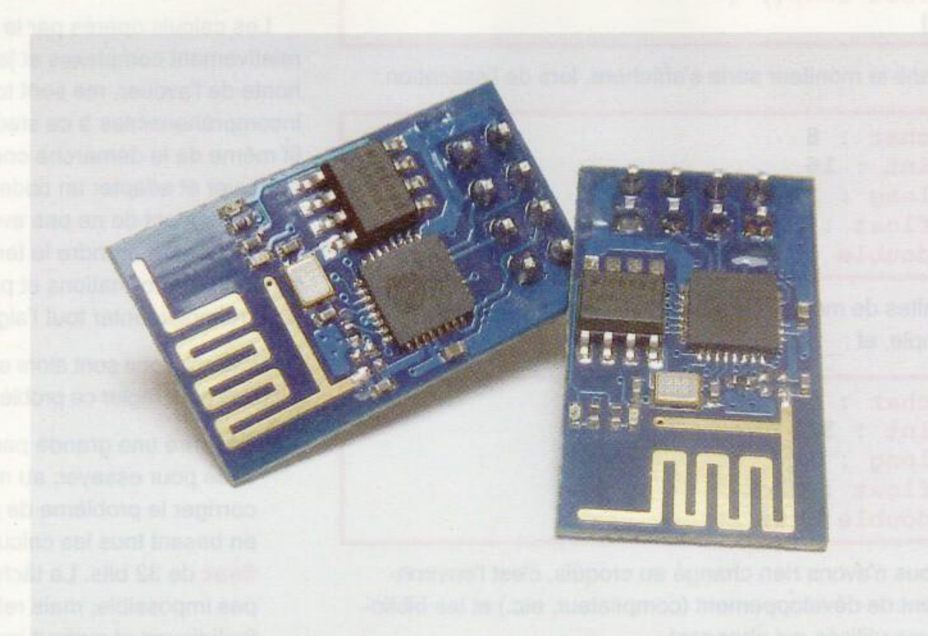
comprendre clairement ce que fait le fameux code. Or justement, c'est précisément ce qu'on essaie d'éviter ici (hé oui, je veux juste une « veilleuse » lunaire, pas créer un planétarium ou me prendre pour Galilée).

- Trouver une solution pour pouvoir utiliser des vrais `double` de 64 bits dans l'environnement Arduino pour Atmel AVR. Là encore, dans l'absolu, c'est quelque chose de parfaitement envisageable... sous réserve d'y mettre le temps et les efforts nécessaires. Il ne s'agit pas seulement de disposer du type `double` ou de trouver une astuce pour « émuler » ce type, mais également d'adapter toutes les fonctions mathématiques utilisées pour travailler avec des valeurs en double précision. Ceci fait, il faudra ensuite, en plus, adapter le croquis en conséquence. La tâche est encore plus pénible que de tout retravailler avec des `float`.
- Opter pour une solution matérielle. Là, deux options existent : utiliser un coprocesseur ou changer de plateforme. La première peut paraître amusante puisqu'il existe en effet des produits comme l'*uM-FPU64* de Micromega se connectant en SPI ou en i2c et fournissant littéralement ce qu'était le 80387 pour l'Intel 80386. Malheureusement, cette solution est coûteuse et impose également une adaptation du code. La seconde solution est bien plus rapide et aisée puisqu'il suffit de trouver une carte avec un vrai `double` de 64 bits et des fonctions mathématiques standards correspondantes.

J'ai insidieusement évoqué cette solution précédemment puisque, en effet, le croquis fonctionne déjà à la perfection avec une carte Arduino Due (ainsi que la Zero ou encore la MKRZero), utilisant un Atmel AT-91SAM3X8E (32 bits) et une chaîne de compilation totalement différente de celle des Arduino AVR.

Le petit problème n'est cependant pas technique, mais financier. Une carte Arduino Due officielle vous coûtera quelques 35€ et un clone environ 15€. Sachant que cette plateforme peut faire bien énormément que de simplement accéder à une horloge en temps réel (RTC) et contrôler quelques leds, cela serait un beau gâchis que de la réserver à ce type d'usage (pas de planétarium, j'ai dit !).

Le but est donc de trouver une plateforme supportant des opérations arithmétiques en double précision la plus économique possible et tout aussi facile à programmer



Le module ESP-01, généralement le moins cher, pourrait convenir si nous nous limitons à l'aspect logiciel. Malheureusement le nombre d'entrées/sorties est trop limité pour contrôler facilement à la fois une horloge et un système de leds (mais ce n'est pas impossible dans l'absolu en utilisant un PCF8574).

qu'une carte Arduino. Idéalement, je dirais même qu'il faudrait que cette carte puisse directement prendre en charge notre croquis sans changements (un bon programmeur est un programmeur feignant) et ce dans l'environnement Arduino... Est-ce beaucoup demander ? Eh bien non, cette plateforme existe et nous en avons déjà parlé dans le magazine, c'est l'ESP8266 !

La plateforme ESP8266 a bien évolué depuis le numéro 7 du magazine où elle était en vedette en couverture. Elle s'est énormément popularisée, le nombre de cartes s'est grandement étoffé et le support dans l'environnement Arduino a gagné en maturité et en stabilité (compilateur, gestion, intégration, bibliothèques, etc.).

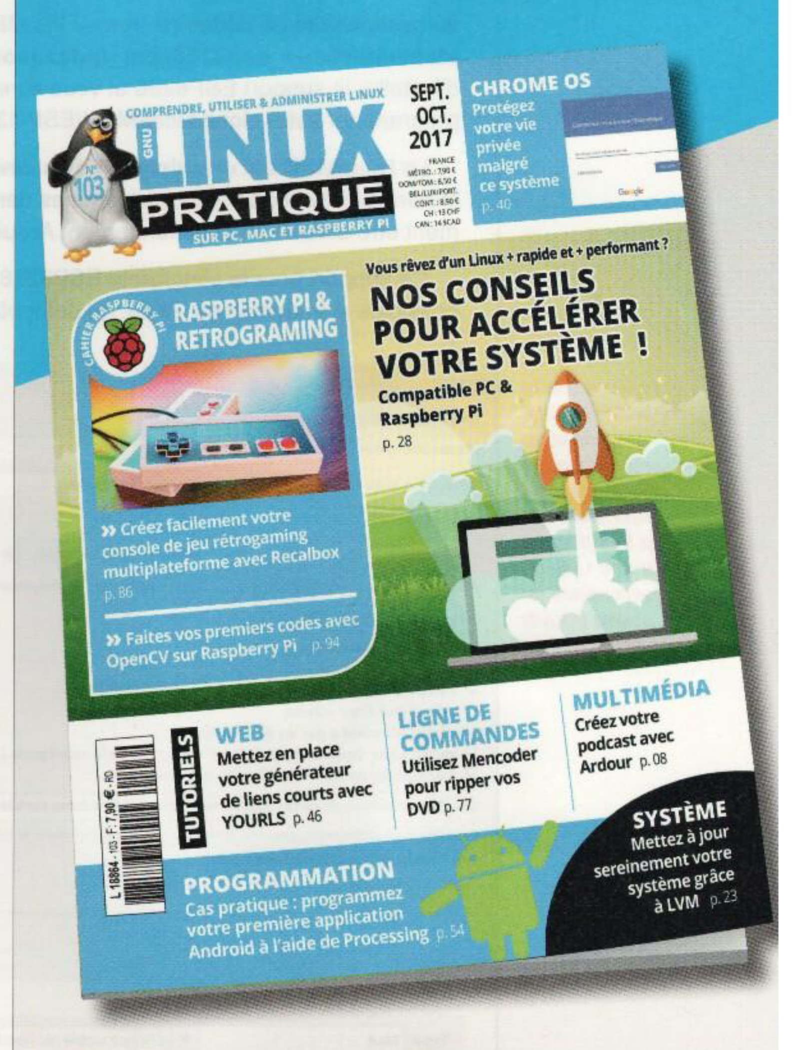
Pour rappel, cette plateforme a commencé sa carrière dans le monde de l'électronique amateur et de la bidouille comme un module de communication permettant à un montage à base d'Arduino, par exemple, de bénéficier d'une connectivité Wifi. Rapidement, de nombreux développeurs se sont rendu compte qu'il était bien plus efficace d'utiliser le module seul et de programmer l'ESP8266 exactement comme le microcontrôleur d'une carte Arduino.

Il existe à présent toute une famille de « modules » ESP8266 du plus simpliste et économique ESP-01, très pauvre en terme de connexions, jusqu'aux modèles très complets montés sur un circuit imprimé fournissant à la fois la connectivité en termes d'entrée/sorties, beaucoup de mémoire et tout le nécessaire pour l'alimentation et la programmation via un port USB. Le plus populaire, à mon avis, est le modèle ESP-12 également utilisé pour la plateforme NodeMCU regroupant tout le nécessaire à l'instar d'une carte Arduino.

Ce type de plateforme se trouve très facilement entre 4€ et 5€ (port gratuit) en cherchant simplement quelque chose comme « *EPS8266 ESP-12 nodeMCU* » sur eBay ou Amazon. Il faudra ensuite se tourner vers son environnement Arduino, et en particulier les préférences, pour ajouter une URL permettant

ACTUELLEMENT DISPONIBLE !

LINUX PRATIQUE n°103



NOS CONSEILS POUR ACCÉLÉRER VOTRE SYSTÈME !

NE LE MANQUEZ PAS CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :



http://www.ed-diamond.com

au gestionnaire de cartes de trouver les éléments à installer : **http://arduino.esp8266.com/stable/package_esp8266com_index.json**. Enfin, un tour dans le gestionnaire de carte afin d'installer le support ESP8266 et vous aurez alors à votre disposition tout le nécessaire pour programmer votre clone NodeMCU/ESP-12 aussi facilement qu'une simple carte Arduino.

Le NodeMCU est une plateforme de développement basée sur l'ESP8266 permettant initialement de développer des programmes dans un langage appelé Lua, mais celle-ci peut également être utilisée via l'environnement Arduino comme nous allons le faire.

Ce support pour la plateforme ESP8266 prend en charge presque tous les modèles possibles et imaginables. Et ce, au point où retrouver le bon modèle dans la liste n'est pas

Paramètres | Réseau
Emplacement du carnet de croquis
/home/denis/sketchbook — Parcourir
Choix de la langue : Langue du système (nécessite un redémarrage d'Arduino)
Taille de police de l'éditeur : 12
Interface scale: ✔ Automatique 100 % (nécessite un redémarrage d'Arduino)
Afficher les résultats détaillés pendant : ✔ compilation ✔ téléversement
Avertissement du compilateur: Rien
Afficher les numéros de ligne
Activer le repli de code
✔ Vérifier le code après téléversement
Utiliser un éditeur externe
✔ Vérifier les mises à jour au démarrage
✔ Mettre à jour l'extension des fichiers croquis lors de la sauvegarde (.pde -> .ino)
✔ Sauvegarder pendant la vérification ou le transfert
URL de gestionnaire de cartes supplémentaires http://arduino.esp8266.com/stable/package_esp8266com_index.json
Davantage de préférences peuvent être éditées directement dans le fichier
/home/denis/.arduino15/preferences.txt
(éditer uniquement lorsque Arduino ne s'exécute pas)
OK Annuler

Pour ajouter le support ESP8266 dans votre environnement de développement Arduino, vous devrez tout d'abord ajouter une URL où le gestionnaire de carte pourra trouver les éléments.

Type Tout — Filtrez votre recherche
EMoRo 2560. Board based on ATmega 2560 MCU.
Online help
More info
AMEL-Tech Boards by **replaced by Arrow Boards**
Cartes incluses dans ce paquet:
SmartEverything Fox.
Online help
More info
esp8266 by **ESP8266 Community** version **2.3.0** INSTALLED
Cartes incluses dans ce paquet:
Generic ESP8266 Module, Olimex MOD-WIFI-ESP8266(-DEV), NodeMCU 0.9 (ESP-12 Module), NodeMCU 1.0 (ESP-12E Module), Adafruit HUZZAH ESP8266 (ESP-12), ESPresso Lite 1.0, ESPresso Lite 2.0, SparkFun Thing, SweetPea ESP-210, WeMos D1, WeMos D1 mini, ESPino (ESP-12 Module), ESPino (WROOM-02 Module), WifInfo, ESPDuino.
Online help
More info
Sélectionner une versi... Installer Supprimer
Fermer

Une fois que l'environnement Arduino sait où retrouver le nécessaire, ajouter le support d'une nouvelle plateforme est un jeu d'enfant puisque ceci se limite à cliquer sur le bouton « Installer » dans le gestionnaire de cartes.

toujours évident. Dans le cas d'un clone de NodeMCU, on hésitera généralement entre l'ESP-12 (NodeMCU 0.9) et l'ESP-12E (NodeMCU 1.0). Visuellement, la différence réside dans le logo dans le coin supérieur gauche du blindage du module : « AI » pour l'ESP-12E et un logo Espressif Inc en forme de boule pour l'ESP-12.

4. AVANT DE PASSER À LA PRATIQUE...

Cet article que je qualifierai de préparatoire montre que la récupération d'un code existant est quelque chose de toujours faisable. Le propre d'un langage comme C/C++, utilisé avec Arduino, est d'être portable. Même si aujourd'hui on pense naturellement à des choses comme Java ou Python pour créer du code portable (facile à adapter sur plusieurs plateformes), le C a été initialement créé justement dans ce sens. Ici, le code « actif » de `moon.c` peut être utilisé sur n'importe quel environnement (Raspberry Pi, PC, console, machine ancienne, cartes, microcontrôleur, etc.) pour peu que l'on prenne en compte ses particularités et celle de l'environnement cible.

Enfin, je soulignerai le fait que récupérer et retravailler un code est, comme vous pouvez le voir, un exercice très enrichissant en plus de permettre de reposer sur le travail et l'expertise de quelqu'un d'autre. Ici le code utilisé était placé dans le domaine public, mais il en va, bien entendu, de même pour le logiciel libre. Il y a sur le Web une quantité astronomique de codes et de morceaux de code qui n'attendent que votre attention pour fonctionner dans vos projets. Si une bibliothèque toute faite n'existe pas, chercher sur d'autres plateformes est souvent une bonne idée avant de vous lancer dans une réécriture complète. Dans les sources du système de votre Raspberry Pi, par exemple, vous avez toutes les chances de trouver votre bonheur, quel que soit votre objectif... **DB**

Besoin d'un port UART, SPI, SWD, JTAG, ou de quelques GPIO ?

Cowstick

- Reconnu comme une carte réseau (Ethernet over USB)
- Serveur WEB embarqué avec API REST
- Bootloader accessible en HTTP

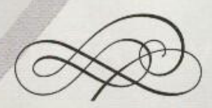




www.cowlab.fr/cowstick

CRÉEZ UNE LAMPE LUNAIRE : EN ROUTE !

Denis Bodor



Dans l'article précédent, nous avons récupéré et réutilisé un morceau de code permettant de faire tous les calculs importants et avons réglé le problème lié au manque de précision dans ces derniers. Nous sommes maintenant prêts pour mettre tout cela en œuvre, utiliser les fonctions à notre disposition et terminer par la construction de notre lune personnelle lumineuse.

Si vous avez sauté directement à cet article (ouh, vilain lecteur, vilain !), nous avons donc à ce stade décidé d'utiliser une carte NodeMCU/ESP8266 pour notre projet, car celle-ci est très économique et l'environnement de développement associé dispose des bibliothèques, fonctions et types de variables dont nous avons besoin. Nous disposons de fonctions nous permettant d'obtenir, à partir d'une date et heure, tout un tas d'informations sur l'état de la lune. Cependant, seules deux informations nous intéressent ici : la partie actuellement éclairée de la lune et éventuellement la date de la prochaine pleine lune (ce n'est qu'une option pour une future évolution du projet).

1. METTONS UN PEU D'ORDRE

La première chose à faire pour intégrer du code « étranger » dans un projet est, à mon sens, de l'isoler sur le reste de notre création. Ceci permet de confiner les éventuels problèmes et de clairement séparer la partie « métier » du reste du projet. Bien entendu, le fait que les fonctions importées soient bien identifiées est une première étape propre au langage utilisé. À ce niveau, l'idée se résume à ne pas polluer un code qui fonctionne très bien et qui est portable (car non spécifique à une plateforme) avec des appels à des fonctions propres à Arduino.

Mais je pousserais le raisonnement encore plus loin au point de limiter ce code d'emprunt à un fichier spécifique prenant la forme d'un onglet dans l'environnement de développement. Le code importé sera donc placé dans un fichier **moonfnct.cpp** (oui, c'est du C, mais dans un fichier **.cpp**, vous allez comprendre). Lorsqu'on travaille ainsi avec plusieurs fichiers, notre croquis **moon** placé dans le premier onglet doit pouvoir savoir que les fonctions présentes dans **moonfnct.cpp** existent.

En effet, le compilateur intégré dans l'environnement transforme les fichiers sources (**.c**, **.cpp**) en fichier objet (**.o**) individuellement. On se retrouve donc avec plusieurs de ces fichiers qui sont ensuite liés entre eux pour former un binaire (au format ELF). Celui-ci est ensuite transformé en représentation hexadécimale pour être chargé dans la mémoire du microcontrôleur. Le compilateur au moment de compiler le croquis n'a aucune connaissance du code présent dans les autres fichiers, ceci c'est l'affaire de l'éditeur de liens créant un binaire.

Ainsi, notre croquis doit avoir un modèle de ce à quoi ressemblent les fonctions des autres fichiers : des prototypes. Comme ces prototypes de fonctions doivent être partagés entre les fichiers sources, on les place généralement dans un fichier d'entête qui est inclus. Dans notre cas, nous allons donc créer un nouveau fichier/onglet, appelé **moonfnct.h**, regroupant les macros utilisées dans le code importé (**#define**), les inclusions de bibliothèques Arduino (pour l'horloge en temps réel) et les prototypes des fonctions se trouvant dans **moonfnct.cpp** dont nous avons parlé dans l'article précédent :

```
// convertir une date en jours juliens en années, mois et jours
void jyear(double td, int *yy, int *mm, int *dd);

// convertir une date en jours juliens en heures, minutes et secondes
void jhms(double j, int *h, int *m, int *s);

// retourne la date de la nouvelle lune
double meanphase(double sdate, double k);

// retourne la date précise pour une phase lunaire donnée
double truephase(double k, double phase);
```

```
// donne des informations sur les prochaines phases lunaires
void phasehunt(double sdate, double phases[5]);

// retourne la date de la prochaine phase spécifiée
double nextmoon(double sdate, double phase);

// calcule l'équation de Kepler
double kepler(double m, double ecc);

// convertit une date standard en jours juliens
double jdate(tmElements_t *t);

// fait de même, mais avec l'heure en plus
double jtime(tmElements_t *t);

// calcul diverses données lunaires
double phase(double pdate, double *pphase, double *mage, double *dist,
double *sudist);
```

Un prototype n'est pas différent d'une déclaration de fonction si ce n'est que le corps de la fonction n'est pas présent et que la déclaration se termine, bien naturellement, par un point-virgule. Ce fichier d'entête sera ensuite inclus dans `moonfnct.cpp` et dans notre croquis avec une ligne `#include "moonfnct.h"`. Notez l'utilisation de guillemets et non de `<` et `>` comme pour une bibliothèque, le fichier d'entête en question est « local ».

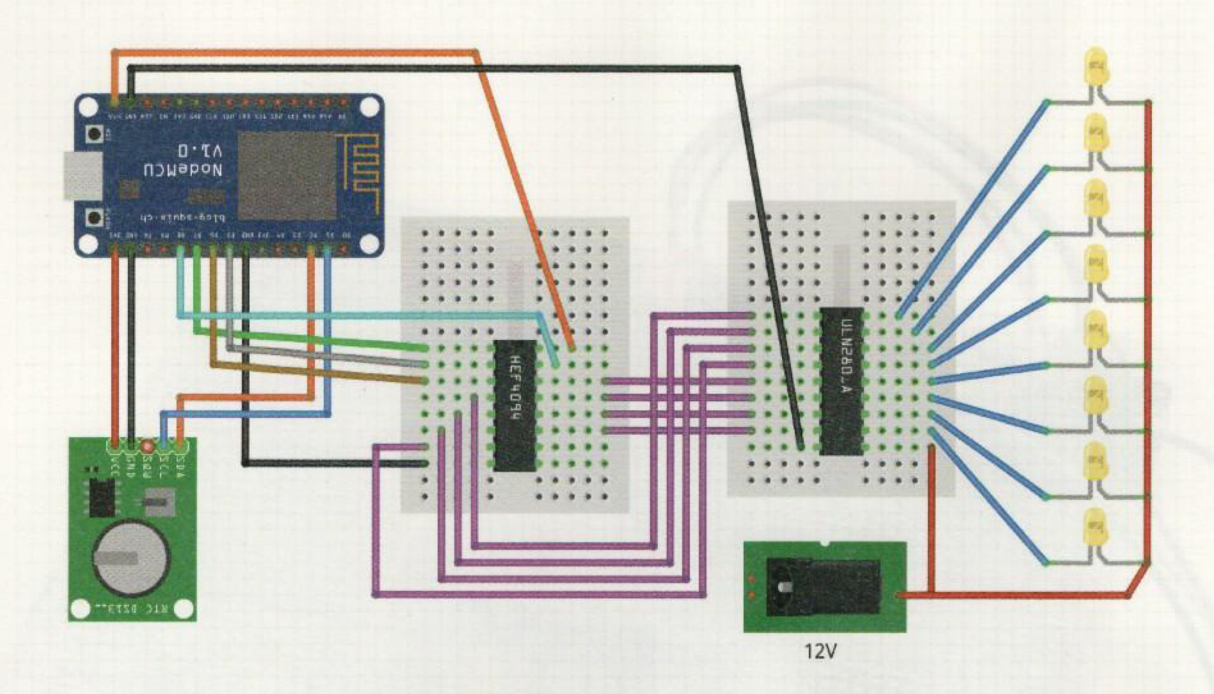
Je vous fais grâce ici de l'ensemble du code importé qui n'apporte pas grand-chose à la thématique de l'article. Celui-ci est disponible dans les fichiers du magazine présent sur **https://github.com/Hackable-magazine/Hackable18**.

2. LE MONTAGE

Le montage complet repose sur une carte NodeMCU, mais n'importe quelle autre carte ESP8266 pourra faire l'affaire du moment qu'elle met effectivement à disposition les sorties adéquates ou, du moins, en nombre suffisant. L'ESP8266 fonctionne en 3,3V, mais est tolérant au 5V (ceci est enfin officialisé après de longs mois de doute). Ceci signifie cependant que nous ne pouvons pas utiliser d'horloge temps réel DS1307, mais devons opter pour une DS1338 pouvant être alimenté entre 3V et 5,5V. Les modules à base de DS3231 pourront également être utilisés, à condition d'adapter légèrement le croquis.

Le DS1338 est un composant interfacé en i2c, mais l'ESP8266 ne dispose pas de bus i2c matériel. L'environnement fournit néanmoins une bibliothèque `Wire` permettant l'utilisation de composants i2c en utilisant deux broches de façon logicielle (*bitbanging*). Les broches par défaut sur une carte NodeMCU sont D2 pour le signal SDA et D1 pour SCL.

Pour contrôler des leds, les choses sont un peu plus délicates. Seront utilisés ici des rubans de leds blanches alimentées en +12V afin d'éclairer la lune de l'intérieur (cf. plus loin dans l'article). Mais le problème n'est pas tant la tension utilisée que le nombre de leds à gérer et donc le courant à manipuler. De plus, l'idée derrière ce projet est également de se laisser l'opportunité de le faire évoluer. L'une des évolutions possibles consisterait, par exemple, à ne pas segmenter la lune en uniquement 8 « quartiers », mais en bien davantage, et ce à une échelle tout autre (une énorme lune de 2m de diagonale par exemple).



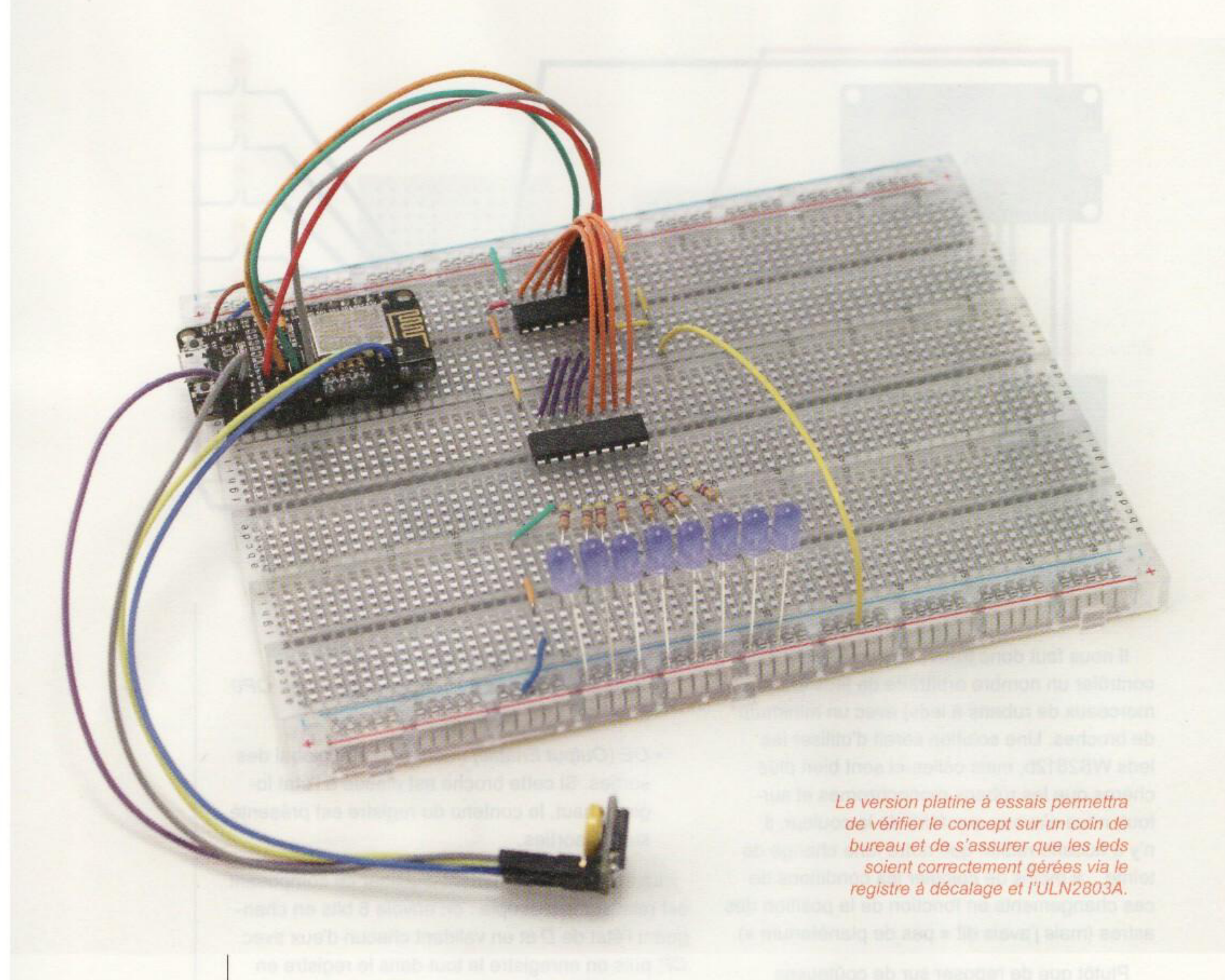
Il nous faut donc trouver un moyen de contrôler un nombre arbitraire de leds (ou de morceaux de rubans à leds) avec un minimum de broches. Une solution serait d'utiliser les leds WS2812b, mais celles-ci sont bien plus chères que les rubans monochromes et surtout, nous n'avons que faire de la couleur, il n'y a aucune raison que notre lune change de teinte... à moins de calculer les conditions de ces changements en fonction de la position des astres (mais j'avais dit « pas de planétarium »).

Plutôt que de reposer sur de coûteuses WS2812b, nous pouvons tout simplement utiliser un registre à décalage comme le 74HC4094. Celui-ci se contrôle généralement via quatre signaux/broches (mais il est possible de se passer de certains d'entre eux) :

- *D* (*Data*) : permet de présenter une donnée binaire (0 ou 1) devant s'ajouter dans le registre.
- *CP* (*Clock Pulse*) : valide la donnée présentée sur *D*. Il s'agit donc d'un envoi en série des 8 bits totaux. On présente un bit et on le valide avec CP.
- *STR* (*Strobe*) : enregistre les données présentées dans le composant. L'état des sorties, déterminées par les 8 bits envoyés est stocké en mémoire et peut alors influer l'état des 8 sorties du composant (notées *QP0* à *QP7*).
- *OE* (*Output Enable*) : contrôle l'état global des sorties. Si cette broche est placée à l'état logique haut, le contenu du registre est présenté sur les sorties.

La technique permettant d'utiliser ce composant est relativement simple : on envoie 8 bits en changeant l'état de *D* et en validant chacun d'eux avec *CP*, puis on enregistre le tout dans le registre en passant *STR* à l'état haut et, éventuellement, on active ensuite les sorties avec *OE*. Il est toutefois possible de physiquement relier à la tension d'alimentation la ligne *OE* afin que les changements d'état soient toujours appliqués sur les sorties « en direct », et nous pourrions économiser une broche.

L'avantage de ce type de registre à décalage est sa modularité. Non seulement nous pouvons utiliser les sorties en 3,3V du NodeMCU pour obtenir des sorties en 5V, mais, de plus, il est possible de chaîner plusieurs 74HC4094 de façon à obtenir 16, 24, 32... sorties, toujours contrôlées par seulement 4 broches. L'objectif ici est de diviser la surface lunaire en 8 segments contrôlables, mais nous pourrions très facilement multiplier cette quantité en utilisant plusieurs registres à décalage chaînés.



La version platine à essais permettra de vérifier le concept sur un coin de bureau et de s'assurer que les leds soient correctement gérées via le registre à décalage et l'ULN2803A.

Enfin, comme les sorties du 74HC4094 ne sont pas en mesure de contrôler directement des rubans de leds alimentés en 12V, celles-ci sont reliées à un ULN2803A. Ce circuit intégré est composé de huit transistors en montage darlington capables d'accepter le courant nécessaire aux leds jusqu'à une tension de 50V. Notez que les ULN2803A ne fournissent pas le courant, l'anode des rubans de leds est donc connectée à la tension d'alimentation et leur cathodes à l'ULN2803A.

Attention : L'utilisation d'un ULB2803A dépendra totalement du nombre de leds utilisées et du courant qui circulera. Dans l'état et avec quelques leds, il est déjà fortement recommandé d'ajouter un dissipateur thermique (radiateur) sur le composant. Si vous comptez utiliser les longs rubans, il vous faudra remplacer l'ULN2803A par autant de MOSFET-N IRL520, par exemple, que vous aurez de segments à contrôler.

Nous avons donc ici, en tout, trois tensions en usage. Les sorties du NodeMCU sont en +3,3V, celles du 74HC4094 en +5V (tension d'alimentation du 74HC4094, fournie par le NodeMCU via la broche *Vin* correspondant à la tension USB) et +12V permettant d'alimenter les leds. Notez que le schéma ci-contre présente les rubans de leds comme de simples leds blanches, mais c'est uniquement pour simplifier visuellement le montage (ne branchez pas des leds au +12V !).

3. LE CROQUIS PRINCIPAL ET NOS FONCTIONS « MAISON »

À présent que nous avons mis un peu d'ordre dans le code et défini l'aspect matériel du projet, il ne nous reste plus qu'à ajouter le code de notre croquis pour lier tout cela.

Commençons par la fonction **setup()** qui nous permet de planter le décor et de détailler ensuite les fonctions utilisées :

```
#include "moonfnct.h"

#define PIN_D   D5
#define PIN_STR D7
#define PIN_CP  D6
#define PIN_OE  D8

#define TAILLELED 8

void setup() {
  // structure pour stocker la date/heure
  tmElements_t tm;

  // configuration des broches
  pinMode(PIN_D, OUTPUT);
  pinMode(PIN_STR, OUTPUT);
  pinMode(PIN_CP, OUTPUT);
  pinMode(PIN_OE, OUTPUT);

  // configuration série
  Serial.begin(115200);
  Serial.println("Go go go");

  // lecture de l'horloge
  if(!RTC.read(tm)) {
    Serial.println("RTC error");
    while(1) {;}
  }

  // petite animation
  animled();
  delay(100);

  // information
  Serial.print("Maintenant (GMT): ");
  printTmDate(tm);

  // affichage illumination et gestion des leds
  barmoon(&tm);

  // affichage prochaine pleine lune
  printnextnew(&tm);

  // affichage prochaine nouvelle lune
  printnextfull(&tm);
}
```

Lors de sa mise sous tension, la carte NodeMCU (ou compatible) tentera de lire la date et l'heure enregistrée dans l'horloge, juste après avoir configuré les broches reliées au registre à décalage. Notez ici que dans le cas d'un DS1338, la fonction RTC.read() retournera une valeur non nulle en cas de problème, mais que, dans le cas d'une DS3231, ce sera l'inverse. Cette fonction est mise à disposition à la fois par les bibliothèques DS1307RTC et DS3232RTC, permettant respectivement la prise en charge des puces DS1338/DS1307 et DS3231. Il faudra donc faire attention à cette ligne selon l'horloge que vous comptez utiliser.

Une fois cet appel fait correctement, tm doit contenir une date et une heure telles que lues dans l'horloge. Nous pouvons ensuite nous en servir pour l'afficher, calculer l'activation des leds puis afficher, sur le moniteur série, les dates et heures des prochaines pleine lune et nouvelle lune.

Pour afficher la date lue, nous construisons une fonction printTmDate() se contentant de prendre les données dans la structure tm passée en argument et à les afficher du mieux possible :

```
void printTmDate(tmElements_t tm) {
  if(tm.Day<10) Serial.print("0");
  Serial.print(tm.Day);
  Serial.print("/");
  if(tm.Month<10) Serial.print("0");
  Serial.print(tm.Month);
  Serial.print("/");
  Serial.print(tm.Year+1970);
  Serial.print(" ");
  if(tm.Hour<10) Serial.print("0");
  Serial.print(tm.Hour);
  Serial.print(":");
  if(tm.Minute<10) Serial.print("0");
  Serial.print(tm.Minute);
  Serial.print(":");
  if(tm.Second<10) Serial.print("0");
  Serial.println(tm.Second);
}
```

Notez que la structure tmElements_t, n'est pas identique à celle que l'on trouve sur Raspberry Pi. L'heure, par exemple, n'est pas stockée dans tm.tm_hour, mais dans tm.Hour. Autre changement notable, l'année est, sur Pi, un nombre d'années écoulées depuis 1900 alors que pour DS1307RTC et DS3232RTC, l'année de base est 1970. Enfin, nous partons du principe, dans l'ensemble du croquis, que l'heure de l'horloge est celle du méridien de Greenwich (GMT) et non l'heure locale (CET ou CEST).

Remarquez que les fonctions jtime() et jdate() utilisent également la structure tmElements_t et un petit ajustement sera nécessaire. De plus, afin d'éviter des inclusions éparpillées dans tous les fichiers du projet, nous avons décidé de placer les #include des bibliothèques Time et DS1307dans notre fichier moonfnct.h. Or DS1307RTC.h est un fichier d'entête C++ (il y a une définition de classe à l'intérieur). Comme DS1307RTC.h est inclus par moonfnct.h, lui-même inclus au début du fichier du code importé, ce dernier ne peut être qu'un code C++ (le compilateur C ne reconnaîtrait pas la directive class de DS1307RTC.h). Ce genre de choses peut être tantôt important, en particulier avec du code importé, certaines choses prennent un sens sensiblement différent en C ou en C++.

Nous pouvons passer à la fonction sans doute la plus importante puisque c'est celle chargée de déterminer l'état des leds, mais également d'afficher une représentation de ces dernières à des fins de mise au point :

```
void barmoon(tmElements_t *tm) {
  // date en jours juliens
  double pdatetime;
  // variables pour phase()
  double pphase, mage, dist, sudist;

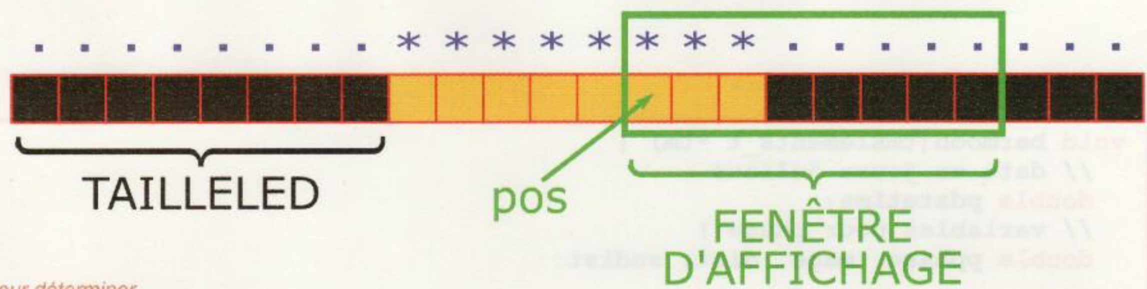
  // construction de la barre
  char moonslide[TAILLELED*3];
  memset(moonslide, '.', TAILLELED*3);
  memset(moonslide+TAILLELED, '*', TAILLELED);

  // conversion de la date
  pdatetime = jtime(tm);

  // calculs
  double pangle = phase(pdatetime, &pphase, &mage, &dist, &sudist);

  // index du premier caractère
  int pos = (int)(pangle*TAILLELED*2);
  // désactivation sorties leds
  digitalWrite(PIN_OE, LOW);
  Serial.print("[");
  // on boucle sur les positions à afficher
  for(int i=pos; i<pos+TAILLELED; i++) {
    Serial.write(moonslide[i]);
    // donnée : "*" ou led allumée
    if(moonslide[i]=='*')
      digitalWrite(PIN_D, HIGH);
    else
      digitalWrite(PIN_D, LOW);
    // clock
    digitalWrite(PIN_CP, HIGH);
    delayMicroseconds(1);
    digitalWrite(PIN_CP, LOW);
    delayMicroseconds(1);
  }
  Serial.println("]");
  // enregistrement dans le registre
  digitalWrite(PIN_STR, HIGH);
  delayMicroseconds(1);
  digitalWrite(PIN_STR, LOW);
  // activation des sorties leds
  digitalWrite(PIN_OE, HIGH);
}
```

L'idée générale derrière cette fonction est la suivante : on compose une « barre » représentant un cycle lunaire complet de nouvelle lune à nouvelle lune comprise, puis on utilise un segment de cette barre pour déterminer l'état des leds. La barre est composée du nombre de leds présentes (**TAILLELED**) multiplié par trois. On détermine le point de départ sur la barre en fonction de l'angle retourné par la fonction **phase()** et on compte toujours le même nombre de positions à partir de cet endroit. Cette incrémentation est également facteur de **TAILLELED**.



Pour déterminer l'illumination de la lune à partir de l'angle retourné par l'une des fonctions originales, on « déroule » simplement le cycle lunaire dans un tableau avec un nombre de cellules correspondant au nombre de leds à contrôler, multiplié par trois. Il suffit alors de déterminer où on commence à compter dans le tableau et lire autant de cellules qu'il y a de leds.

Pour créer cette barre en mémoire, qui en fait est un tableau, de manière à ne pas avoir à changer le code si nous pilotons davantage de leds, nous utilisons également la macro **TAILLELED**. Nous remplissons ensuite le tableau avec des caractères **"."** et **"*"** en utilisant la fonction **memeset()** prenant en argument un emplacement mémoire (un pointeur), un octet et une quantité de mémoire à traiter à partir de l'endroit pointé. Ici nous remplissons d'abord tout le tableau de **"."** puis la partie centrale de **"*"**. Si **TAILLELED** est égal à 8, alors **moonslide[]** contiendra **"........********........"** (sans **\0** puisqu'il ne s'agit pas d'une chaîne de caractères).

Une fois ce tableau rempli dynamiquement, il nous suffit de trouver où commencer la lecture parmi les 16 premières cellules (**TAILLELED** fois 2 donc) et ceci est possible grâce à la valeur de l'angle retournée, entre 0 et 1, par **phase()**. Il ne nous reste plus ensuite qu'à comparer un nombre de cellules correspondant à **TAILLELED** et le tour est joué, à la fois pour l'affichage dans le moniteur série et pour le registre à décalage où l'état de **D** dépendra de la présence de **"*"** dans la cellule lue.

Et enfin nous avons la recherche de la prochaine date de pleine lune et de la nouvelle lune avec des fonctions comme celle-ci :

```
void printnextfull(tmElements_t *tm) {
  // jour, mois, année, heure, minute, seconde
  int dd, mm, yy, hh, mn, ss;

  // date en jours juliens
  double pdatetime = jtime(tm);

  // calcul de la date pour la phase
  double next = nextmoon(pdatetime, 0.5);

  // décodage de la date à partir des jours juliens
  jyear(next, &yy, &mm, &dd);
  jhms(next, &hh, &mn, &ss);

  // affichage
  Serial.print("Prochaine pleine lune = ");
  printDate(dd,mm,yy,hh,mn,ss);
}
```

On utilise simplement la fonction **nextmoon()** décrite précédemment et placée dans le fichier **moonfnct.cpp** puis on traite les données obtenues pour les utiliser avec **printDate()** qui n'est autre qu'une variation de **printTmDate()** prenant en argument différentes valeurs plutôt qu'une structure **tmElements_t**.

Bien entendu, s'il s'agit de suivre en temps réel l'évolution du cycle lunaire, on accompagnera le croquis de la fonction **loop()** reprenant une partie de ce que nous venons de voir pour **setup()**, mais ajoutant une temporisation entre les relevés de l'horloge :

```
void loop() {
  tmElements_t tm;

  // lecture de l'horloge
  if(!RTC.read(tm)) {
    Serial.println("RTC error");
    while(1) {;}
  }

  // information
  Serial.print("Maintenant (GMT): ");
  printTmDate(tm);

  // affichage illumination et gestion des leds
  barmoon(&tm);

  delay(5000);
}
```

4. C'EST LE MOMENT DE BRICOLER !

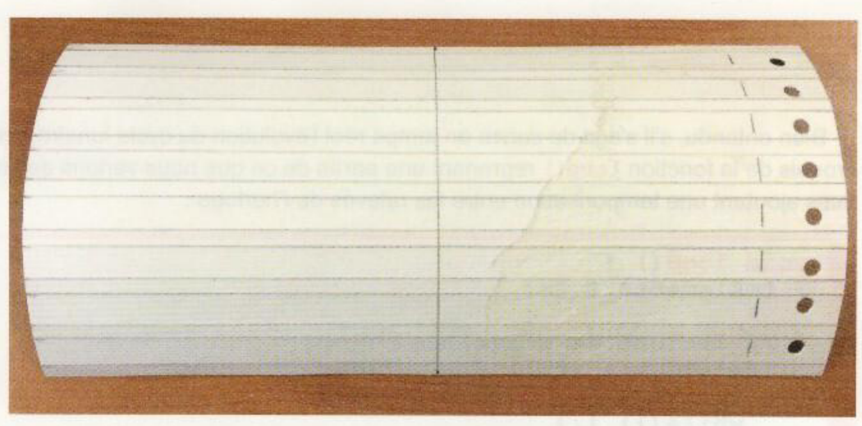
Nous avons le code, nous avons l'électronique, il ne nous reste plus qu'à mettre tout cela en forme pour faire quelque chose de (vaguement) présentable. Je dois avouer que je ne suis pas aussi débrouillard dans le thème du bricolage façon « loisir créatif » que je peux l'être éventuellement dans d'autres domaines, même si l'intention et la motivation sont bien là.

L'idée générale du « bricolage du dimanche » en question est la suivante : utiliser une balle gonflable (style balle de plage) comme support de départ puis la découper pour en faire une hémisphère « molle » qui couvrira un support.

Le support quant à lui sera construit avec du carton-plume de manière à former des quartiers circulaires collés sur une base. Sur cette base seront également collées les lamelles séparant les quartiers et, entre elles, seront placés les rubans de leds contrôlés par le montage.



Bonne idée ou non, mon idée initiale était de tout simplement utiliser une balle/lune gonflable de 26cm de diamètre pour obtenir la texture souhaitée. Ce modèle précis disposait même d'un orifice à la base pour y glisser une lumière, mais c'était insuffisant pour autre chose qu'un éclairage global.



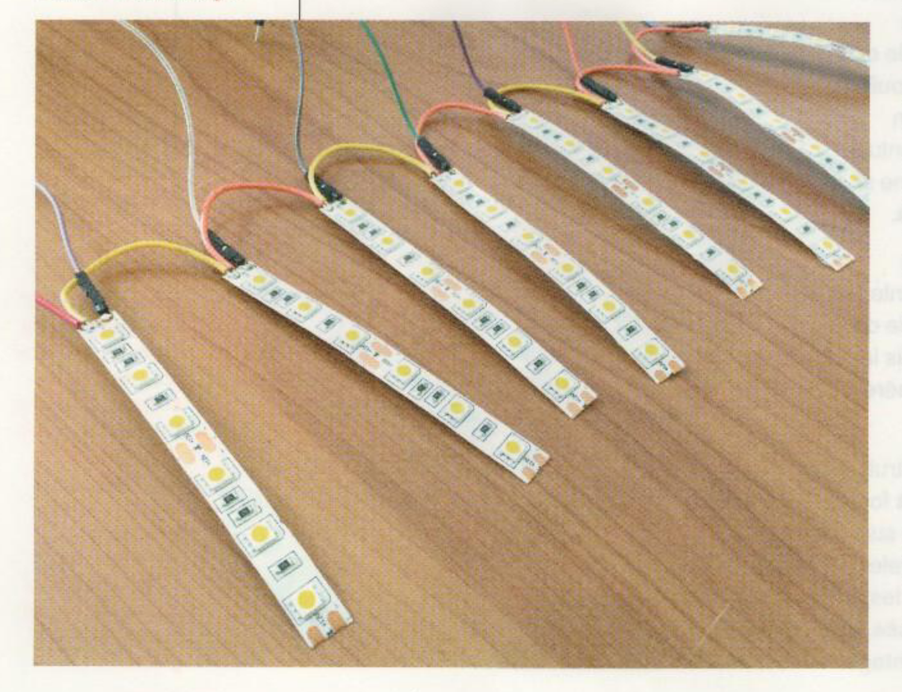
La base de la structure est un simple morceau de carton de récupération avec, tracés au crayon, les emplacements pour les segments et les rubans de leds.

En termes de construction, la partie la plus délicate est d'estimer correctement le rayon de la balle, car celui-ci servira à tracer les formes au compas sur le carton-plume en ayant préalablement retranché le rayon du morceau de carton formant la base de la structure. Les rubans de leds font 1 centimètre de large et le carton-plume 4 mm d'épaisseur. Nous avons 8 phases et donc 9 séparations. Ces mesures sont reportées sur un morceau de carton souple et les « tranches » tracées au crayon. On prendra soin éventuellement de bien repasser sur les traits au stylo à bille pour marquer le support et faciliter la pliure.

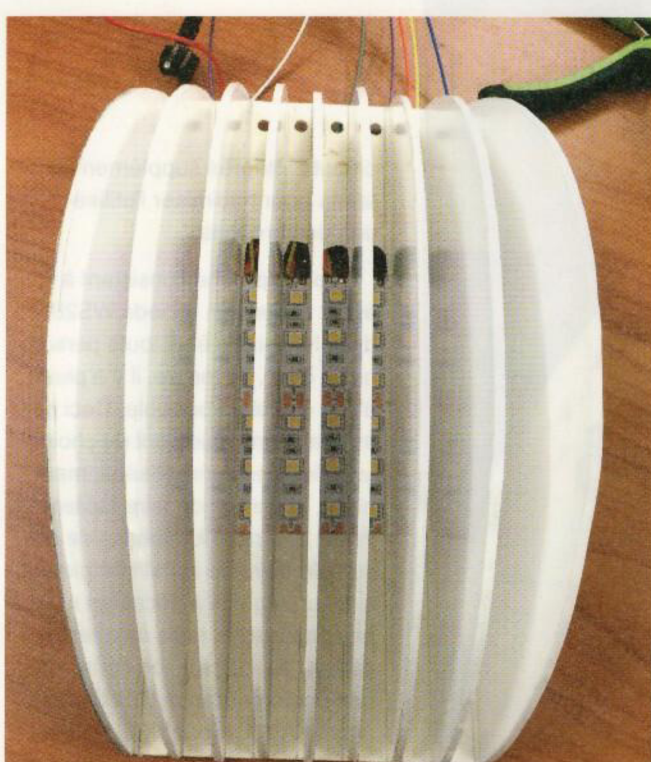
Après plusieurs essais et tâtonnements, on arrive à un équilibre correct et on peut débuter le collage au pistolet à colle. Cet outil est parfaitement adapté, en particulier en utilisant un grand modèle avec des bâtons de 11mm (merci au passage à Sylviane pour le prêt de l'engin). La colle est moins facile à doser, mais la température est idéale pour ce type de support.

Les lamelles en carton plume ainsi collées forment une sorte d'accordéon/éventail qu'on pourra déplier pour former une demi-sphère grossière. Les segments de ruban possèdent généralement un adhésif double face permettant le collage au milieu de chaque quartier. J'ai trouvé plus pratique de souder les anodes des rubans entre elles ainsi que les câbles de connexion avant collage, en pratiquant des ouvertures au cutter aux endroits stratégiques sur la base du support. L'idée de jongler avec le fer à souder entre les morceaux de carton-plume ne me semblait pas très pratique, pas plus que de tenter de placer les lamelles après les leds.

Le câblage préalable des segments de ruban de leds permet de vérifier leur fonctionnement et, en même temps, de faciliter l'assemblage.



On peut alors tester le fonctionnement en plaçant, par dessus, la demi-sphère souple obtenue à partir de la balle. Le résultat que j'ai obtenu n'est pas optimal, car la matière utilisée est destinée à être tendue par la pression de l'air. En l'absence de gonflage, la courbure



Mis à plat après collage des séparations, nous obtenons nos huit phases lunaires individuellement activables. Le collage au pistolet à colle semble fournir une adhésion solide et durable.

n'est pas parfaite et, de plus, les soudures des différentes parties de la surface sont bien trop visibles. Le résultat est un peu difforme, mais acceptable... de loin.

Une autre approche possible sera de travailler en deux dimensions à partir d'une photo de la lune, en plaçant les leds à l'arrière. On obtiendrait alors un disque lunaire plus facile à travailler, mais les leds seraient, quant à elles, plus difficiles à placer pour obtenir un rendu réaliste (il faudrait suivre les courbes des phases de la lune avec des rubans relativement rigides).

Enfin, on peut également acheter une lune en plastique, comme un globe terrestre, qu'on pourra ensuite massacrer à loisir. Le coût cependant sera plus important, car ici la balle et le carton-plume sont l'affaire d'une douzaine d'euros seulement.

En « dépliant » la structure, on obtient précisément le résultat attendu : une demi-sphère segmentée en mesure de diffuser une lumière parfaitement diffuse et contrôlée.





Le résultat final avant collage de la surface n'est pas parfait et mérite encore un peu de travail. Lorsque c'est la pleine lune, par exemple, on distingue clairement les segments, tout autant que les différents morceaux qui composent la surface initiale de la balle. Il n'est pas impossible que je finisse par opter pour une autre approche et revoie ma copie dans un proche avenir.

5. PERSPECTIVES

Ce croquis comme le montage lui-même est prévu pour être facile à étendre et complété par différentes fonctionnalités. Il n'est, par exemple, pas difficile d'imaginer multiplier les leds et donc le nombre de segments de lune contrôlables. Dans l'état, il suffit d'ajouter un ou plusieurs registres à décalage chaînés et d'ajuster `TAILLELED` en conséquence...

De la même manière, les fonctions `printnextfull()` et `printnextnew()` ne font qu'envoyer du texte sur le moniteur série. On pourra rapidement ajouter un afficheur LCD alphanumérique (type HD44780) pour présenter ces informations. Bien entendu, il faudra utiliser un afficheur avec une interface i2c ou jongler avec des circuits intégrés supplémentaires pour minimiser l'utilisation des broches.

Mon approche consistant à éviter l'utilisation de leds WS2812b dites Neopixels était toute personnelle, mais, là encore, il y a plus d'une évolution possible. Ceci permettrait non seulement de choisir la ou les couleurs utilisées, mais aussi, et surtout de faire facilement varier l'intensité lumineuse de l'ensemble (faire de la PWM logicielle avec un registre à décalage n'est pas impossible, mais cela reste délicat). Si, de plus, on connecte une photorésistance (LDR) sur la broche A0 du NodeMCU, il sera même possible d'ajuster automatiquement la luminosité en fonction de l'éclairage d'une pièce. Idéal pour une chambre...

Enfin, la fonctionnalité principale généralement recherchée lorsqu'on en arrive à opter pour l'ESP8266 est ici laissée totalement de côté : le Wifi. Il y a suffisamment d'espace en flash et en RAM pour prévoir, par exemple, un serveur web embarqué présentant différentes informations lunaires en temps réel, voire permettant différents réglages ou encore la modification de la date et heure. On pourra aussi totalement oublier la RTC et récupérer l'information temporelle sur le net auprès d'un serveur NTP ou autre.

Ce qui aura commencé comme un simple projet de décoration pourrait donc se transformer rapidement en quelque chose de plus complexe, voire de totalement différent. Mais cela sera laissé à votre imagination et à votre envie de torturer le concept initial... **DB**

SUMOBOT



ESIEE PARIS, NOISY LE GRAND
10 MARS 2018

CONSTRUIS TON ROBOT ET ENTRE DANS L'ARENE !

Compétition ouverte
à **tous les niveaux**
3 Catégories :
Débutant, **Expert** et **Par Binome**

CODE PROMO :
HACKABLE



MONTE TOI-MEME TON ROBOT ET PREPARE LE POUR LA
1ère **COMPETITION DE ROBOTS SUMO DE FRANCE** GRACE AU KIT
LUDIQUE ET ACCESSIBLE A TOUS LES NIVEAUX !
KIT EN VENTE A PARTIR DE 55 €
DISPONIBLE SUR

www.esieespace.fr / contact@esieespace.fr





THALES





ESIEE
PARIS



VISUALISEZ LA CHARGE PROCESSEUR DE VOTRE PI AVEC PIMORONI PIGLOW

Yann Morère



La petite carte Piglow de chez Pimoroni [1] embarque 18 leds de couleurs pilotables individuellement. Elle peut être utilisée pour réaliser une ambiance lumineuse autour de votre Raspberry Pi, mais aussi pour vous donner visuellement des informations sur votre système. Ainsi, dans cet article nous allons la mettre en œuvre pour notifier la charge CPU globale de votre framboise sur Raspbian et Recalbox.

Le module Piglow est une carte petit format (36mm x 34mm x 7mm) fabriquée par la société Pimoroni pour la Raspberry Pi. Elle est composée de 18 leds de couleurs contrôlables individuellement. Il ne s'agit pas de leds RGB, mais de 3 séries de 6 leds de couleurs différentes disposées en spirale (cf. figure 1). Du centre de la spirale vers l'extérieur, les couleurs sont disposées comme suit : blanc, bleu, vert, jaune, orange et rouge.

Le circuit intégré responsable de la gestion des leds est le SN3218, un microchip PWM 8-bits à 18 canaux. Il communique avec le Raspberry Pi par l'intermédiaire du bus I2C, à l'adresse 0x54. Les leds peuvent se voir attribuer une valeur comprise entre 0 et 255 pour régler leur intensité.

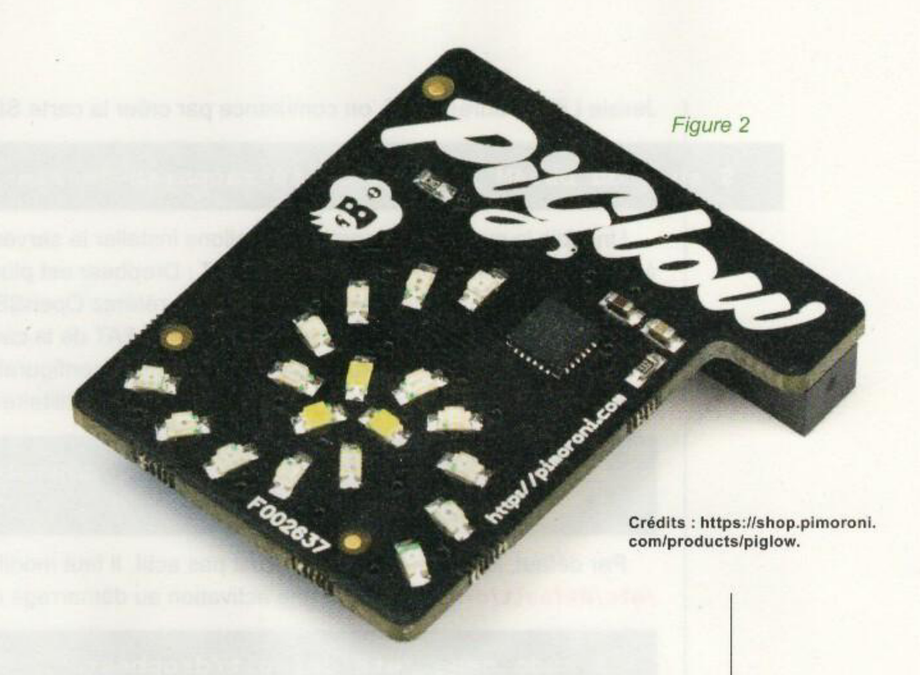
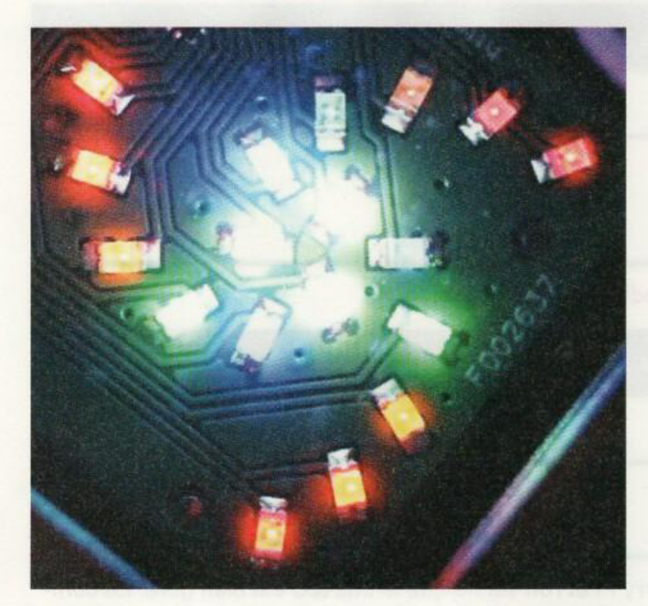


Figure 2

Crédits : https://shop.pimoroni.com/products/piglow.

Prévue au départ pour le Raspberry Pi de première génération, elle possède un connecteur 26 broches. Cependant elle est utilisable avec toutes les versions de Raspberry Pi (cf. figure 2).

Son installation est des plus simples, il suffit de l'enficher sur le GPIO de votre Raspberry Pi. Si vous désirez déporter l'installation de cette carte, le câblage utile est disponible à l'adresse **[2]**.


Figure 1

1. MISE EN ŒUVRE AVEC RASPBIAN

La manière la plus simple de mettre en œuvre le module piglow est d'utiliser la bibliothèque écrite en python par les fabricants de la carte **[3]**. Cependant, elle peut être utilisée avec de nombreux autres langages/bibliothèques : Scratch, WiringPi, Node.js, Golang, Java.

Dans la suite, nous utiliserons la bibliothèque objet **[4]** à la place de la bibliothèque originale. Mais avant cela, il faut installer quelques dépendances sur notre distribution fraîchement installée.

Après avoir téléchargé la dernière version de la distribution Raspbian

Jessie Lite à l'adresse **[5]**, on commence par créer la carte SD :

```
$ sudo dd bs=4M if=2017-04-10-raspbian-jessie-lite.img of=/dev/sdc
```

Une fois le système démarré, nous allons installer le serveur SSH Dropbear pour interagir à distance avec notre Raspberry Pi (NDLR : Dropbear est plus léger que le serveur OpenSSH installé (mais non activé) par défaut. Si vous préférez OpenSSH, ajoutez simplement un fichier `ssh` avec un contenu quelconque sur la partition FAT de la carte SD avant de démarrer votre Pi, le serveur SSH sera alors automatiquement activé (configuration headless)). On en profite pour installer le clavier français pour plus de facilité et des utilitaires pour le bus I2C :

```
$ sudo apt-get install dropbear console-data i2c-tools
$ sudo loadkeys fr
```

Par défaut, le serveur Dropbear n'est pas actif. Il faut modifier son fichier de configuration `/etc/default/dropbear` pour une activation au démarrage du système :

```
$ sudo nano /etc/default/dropbear
```

et on le modifie comme suit :

```
# disabled because OpenSSH is installed
# change to NO_START=0 to enable Dropbear
NO_START=0
```

Ensuite, il s'agit activer le support du bus I2C par le noyau. Pour cela, il faut modifier le fichier `/boot/config.txt` comme suit :

```
$ sudo nano /boot/config.txt
```

et l'on ajoute :

```
#activate i2c
dtparam=i2c1=on
dtparam=i2c_arm=on
```

Ensuite on modifie le fichier `/etc/modules.conf` :

```
$ sudo nano /etc/modules.conf
```

et l'on y ajoute :

```
i2c-bcm2708
i2c-dev
```

On redémarre ensuite notre Raspberry Pi et l'on vérifie que le bus I2C est bien géré. La commande suivante permet de détecter les périphériques I2C :

```
$ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Bien que le module Piglow soit installé et utilise l'adresse 0X54, il n'apparaît pas, cependant il fonctionnera très bien.

On installe ensuite la bibliothèque dédiée à notre petite carte par l'intermédiaire du dépôt git :

```
$ sudo apt-get install git
$ apt-get install i2c-tools python-smbus python-sn3218
$ git clone https://github.com/Boeeerb/PiGlow.git
```

Tout est prêt pour pouvoir utiliser votre piglow. On peut tester l'un des exemples fournis avec la bibliothèque. Par exemple, le programme **cpu.py** permet d'afficher en temps réel la charge cpu de votre Raspberry Pi en allumant un certain nombre de leds avec plus ou moins d'intensité.

Afin de pouvoir utiliser ce programme, il est nécessaire d'installer des utilitaires supplémentaires :

```
$ sudo apt-get install python-psutil
```

On lance le programme dans une première console à l'aide la commande :

```
$ sudo python cpu.py
```

Note : Il est nécessaire d'être administrateur afin de bénéficier des droits requis pour l'utilisation du bus I2C.

Pour tester le programme avec le cpu en charge, on pourra utiliser le programme **sysbench** dans une autre console :

```
$ sudo apt-get install sysbench
```

Il s'utilise comme suit :

```
$ sysbench --test=cpu --cpu-max-prime=20000 run
```

pour un système mono cœur.

Dans le cas d'un Raspberry Pi 3, il peut être intéressant d'utiliser l'option **--num-threads** afin de mettre en charge tous les cœurs du cpu.

```
$ sysbench --test=cpu --cpu-max-prime=20000  --num-threads=4 run
```

Ceci permet de tester différents niveaux de charges CPU et le comportement associé du programme **cpu.py** en changeant le nombre de threads utilisés (de 1 à 4).

2. L'API PYTHON DE PIGLOW

La classe PiGlow ne contient que 14 méthodes qui permettent de contrôler complètement le module.

Votre programme python devra toujours commencer par l'importation de la classe et l'instanciation d'un objet de type PiGlow :

```
from piglow import PiGlow
piglow = PiGlow()
```

Cet objet sera ensuite utilisé pour accéder au pilotage des leds.

Ces dernières sont adressables soit par leur couleur, soit par leur numéro. L'intensité d'éclairage est réglable par une valeur entière comprise en 0 et 255. Elles peuvent aussi être pilotables par groupe de 6, car elles sont disposées sur 3 « bras » en spirale.

Le tableau ci-dessous résume l'ensemble des méthodes utilisables :

Méthodes	Options/Définition
piglow.colour([1-6],[0-255])	1=Blanc, 2=Bleu, 3=Vert, 4=Jaune, 5=Orange, 6=Rouge
piglow.colour([color],[0-255])	"white", "blue", "green", "yellow", "orange", "red"
piglow.white([0-255])	Contrôle toutes les leds blanches
piglow.blue([0-255])	Contrôle toutes les leds bleues
piglow.green([0-255])	Contrôle toutes les leds vertes
piglow.yellow([0-255])	Contrôle toutes les leds jaunes
piglow.orange([0-255])	Contrôle toutes les leds oranges
piglow.red([0-255])	Contrôle toutes les leds rouges
piglow.all([0-255])	Contrôle toutes les leds
piglow.led([1-18],[0-255])	Contrôle une led individuellement par son numéro
piglow.led1-led18([0-255])	Contrôle une led individuellement par sa propre fonction
piglow.arm([1-3],[0-255])	Contrôle un bras de leds du module via son numéro
piglow.arm1([0-255])	Contrôle le bras supérieur
piglow.arm2([0-255])	Contrôle le bras de droite
piglow.arm3([0-255])	Contrôle le bras de gauche

L'étude des exemples fournis avec la classe vous permettra facilement d'écrire vos propres programmes. La page **[4]** présente aussi une méthode adaptée à Raspbian pour lancer au démarrage un programme python qui utilise le module PiGlow.

Voyons maintenant comment l'intégrer à la distribution de retrogaming Recalbox pour avoir une information lumineuse en temps réel correspondant à la charge du système pendant son exécution.

3. MISE EN ŒUVRE AVEC RECALBOX

Recalbox **[6]** n'est pas basée sur une distribution Linux à l'instar de Retropie qui utilise Raspbian par exemple. Elle se base sur Buildroot qui permet de générer une distribution « Linux from scratch » optimisée en fonction de l'architecture et possédant les seuls paquets nécessaires à la destination du système. Ainsi, dans Recalbox, il n'existe pas de gestionnaire de paquets pour l'installation d'applications supplémentaires. Seuls les logiciels voulus par l'équipe de développement seront présents. Ainsi il ne sera pas aisé d'ajouter facilement de nouvelles fonctionnalités sans passer par le développement d'un « package » à intégrer dans le système Buildroot de Recalbox. Je vous renvoie à la lecture de l'article dédié à Recalbox dans le hors-série *Linux Pratique* n° 38 pour une présentation détaillée.

Dans la suite, je considère que votre système Recalbox est opérationnel, enregistré sur votre réseau et que vous pouvez vous y connecter via SSH.

Par chance, Recalbox utilise python pour sa gestion interne, et de nombreux utilitaires et bibliothèques sont déjà présents sur le système. Pour mettre en œuvre notre module PiGlow, il ne sera pas nécessaire de réaliser un développement conséquent, mais seulement d'ajouter quelques fichiers et d'en modifier certains.

Dans la suite, je vais prendre l'exemple d'un petit boîtier de retrogaming à la forme d'une NES : pitendo. Ce boîtier est disponible sur « thingiverse » à l'adresse **[7]**. Ce dernier embarque 2 boutons, qui permettent de réinitialiser le Raspberry Pi, mais aussi de l'éteindre proprement (cf. figure 3). Ces boutons sont directement câblés sur le GPIO et sont gérés nativement par Recalbox.

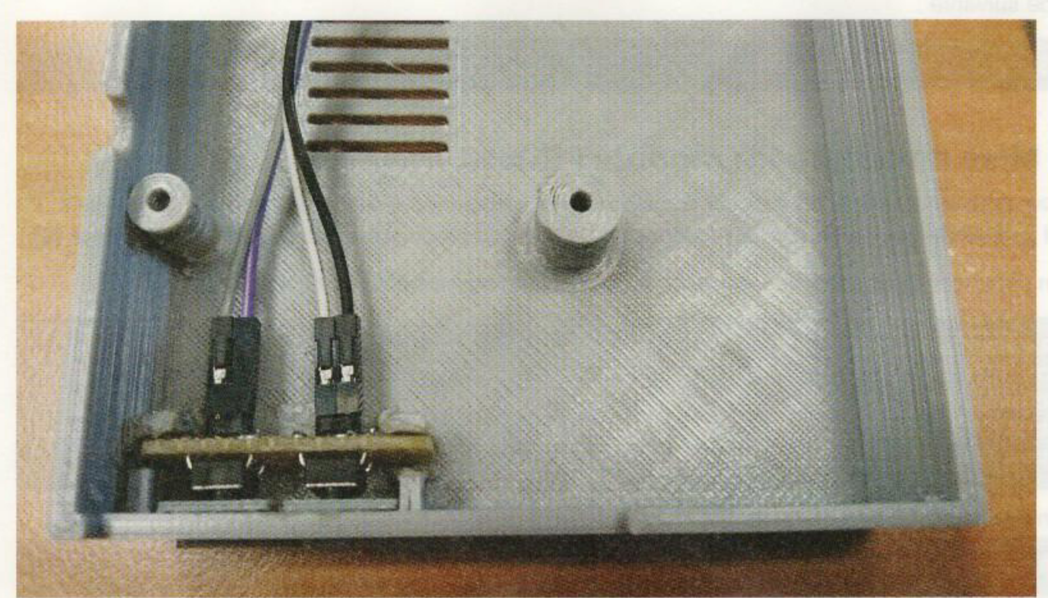


Figure 3

Cependant, le câblage natif utilisé par les scripts utilise les GPIO dédiés au bus I2C qui nous est nécessaire (GPIO 2 et 3 sur les pins 3 et 5), cf. **[8]**. Il nous faudra donc modifier les scripts de Recalbox pour utiliser d'autres GPIO pour les boutons et ainsi libérer le bus I2C.

3.1 Mise en œuvre du bus I2C sur Recalbox

On commence par activer le bus I2C sur le système en modifiant le fichier `/boot/config.txt` comme suit.

```
#activate i2c
dtparam=i2c1=on
dtparam=i2c_arm=on
```

Pour cela, on se connecte en ssh sur le système à l'aide de l'identifiant `root` et du mot de passe `recalboxroot`.

```
$ ssh root@IPdeRecalbox
```

Un fois connecté, il est possible de passer temporairement les partitions en lecture/écriture pour réaliser les modifications par les commandes suivantes :

```
# mount -o remount, rw /boot
# mount -o remount, rw /
```

On modifie ensuite le fichier `/etc/modules.conf` en ajoutant :

```
i2c-bcm2708
i2c-dev
```

On redémarre le système et on vérifie si le bus i2C est bien actif par l'intermédiaire de la commande suivante :

```
$ sudo i2cdetect -y 1
```

3.2 Mise en œuvre du module PiGlow

Pour pouvoir utiliser le module sur Recalbox, il est nécessaire d'ajouter la classe `piglow`, mais aussi la bibliothèque de support du composant `sn3218`. Cette dernière est disponible à l'adresse **[9]**.

On commence par créer un répertoire, puis on télécharge les fichiers nécessaires :

```
# mkdir piglow
# cd piglow
# wget https://github.com/pimoroni/sn3218/blob/master/library/sn3218.py
# wget https://github.com/pimoroni/piglow/blob/master/library/piglow.py
# wget https://github.com/Boeeerb/PiGlow/blob/master/Examples/cpu.
```

On peut alors tester rapidement le fonctionnement de notre module par la commande suivante :

```
# python ./cpu.py
```

Normalement, le module devrait s'allumer et donner une image de la charge cpu de votre système.

Lors de mes essais, les réglages par défaut du fichier **cpu.py** ne sont pas adaptés pour un Raspberry Pi3 et « l'animation » de la charge CPU est assez pauvre en couleur. Après quelques essais, j'ai modifié le programme comme suit (cf. figure 4) :

```
###########################################################
## Show your current CPU usage on your PiGlow!           ##
##                                                       ##
## Requires psutil - sudo apt-get install python-psutil ##
##                                                       ##
## Example by Jason - @Boeeerb   modification ian57     ##
###########################################################

from piglow import PiGlow
from time import sleep
import psutil

piglow = PiGlow()
piglow.all(0)

while True:
    cpu = psutil.cpu_percent()
    piglow.all(0)

    if cpu < 8:
        piglow.red(5)
    elif cpu < 14:
        piglow.red(10)
        piglow.orange(40)
    elif cpu < 22:
        piglow.red(20)
        piglow.orange(60)
        piglow.yellow(60)
    elif cpu < 27:
        piglow.red(30)
        piglow.orange(80)
        piglow.yellow(80)
        piglow.green(80)
    elif cpu < 32:
        piglow.red(40)
        piglow.orange(100)
        piglow.yellow(100)
        piglow.green(100)
        piglow.blue(100)
    elif cpu < 36:
        piglow.red(40)
        piglow.orange(100)
        piglow.yellow(100)
        piglow.green(100)
        piglow.blue(100)
```



Figure 4

```
            piglow.white(100)
        elif cpu < 40:
            piglow.all(120)
        elif cpu < 50:
            piglow.all(140)
        elif cpu < 60:
            piglow.all(160)
        elif cpu < 70:
            piglow.all(180)
        elif cpu < 80:
            piglow.all(200)
        elif cpu < 90:
            piglow.all(220)
        else:
            piglow.all(250)
        sleep(0.1)
```

J'ai aussi réduit la temporisation afin que l'affichage soit plus réactif.

3.3 Mise en route au démarrage

Afin que notre programme d'affichage de charge CPU fonctionne dès le démarrage du système, nous allons créer le fichier **/etc/init.d/S99piglow** contenant le script suivant :

```
#!/bin/sh

case "$1" in
        start)
                printf "Starting piglowcpu ... "
                # Wait for emulationstation or Kodi, but not more than 20 seconds
                count=0
                while [[ ! -f "/tmp/emulationstation.ready" && ! -e "/var/run/
kodi.msg" && $count -lt 20 ]]; do
                        sleep 1
                        ((count++))
                done
                start-stop-daemon -S -q -m -b -p /var/run/piglowcpu.pid --exec
python /recalbox/share/system/piglowcpu/cpu2.py -- -n
                #python /recalbox/share/system/piglowcpu/cpu.py
                echo "done."
                ;;
        stop)
                printf "Stopping piglowcpu ..."
                python /recalbox/share/system/piglowcpu/alloff.py
                start-stop-daemon -K -q -p /var/run/piglowcpu.pid
                echo "done."
                ;;
        restart)
```

```
                $0 stop
                sleep 1
                $0 start
                ;;
    *)
                echo "usage:  $0 {start|stop|restart}"
                ;;
esac
```

Celui-ci fait intervenir les fichiers **cpu.pu** et **alloff.py**. Ce dernier permet d'éteindre toutes les leds de la carte à l'extinction du système :

```
####################
## All Off        ##
##                ##
## Ian57          ##
####################

from piglow import PiGlow
piglow = PiGlow()
piglow.all(0)
```

Pour une bonne intégration à Recalbox, il faut retarder la mise en route du programme qui utilise la carte piglow. En effet si le programme de gestion d'affichage de la charge CPU est lancé trop tôt, il se termine prématurément. Une interaction avec un autre programme doit être à l'origine de ce dysfonctionnement.

Il est donc lancé en dernier (S99) depuis **/etc/init.d** et son activation est retardée de 20s maximum grâce aux lignes suivantes :

```
while [[ ! -f "/tmp/emulationstation.ready" && ! -e "/var/run/kodi.msg" &&
$count -lt 20 ]]; do
            sleep 1
            ((count++))
done
```

Le démarrage et l'arrêt de notre programme d'affichage de charge CPU sont maintenant automatisés sur Recalbox.

3.4 Mise en œuvre des boutons du boîtier Pitendo

Comme indiqué précédemment, la gestion native de ces boutons entre en conflit avec le bus I2C. Il est donc nécessaire de déplacer le câblage de ces derniers vers des broches GPIO non utilisées.

Le support de ces boutons par Recalbox est assuré par la configuration d'une variable dans le fichier **recalbox.conf** :

```
# ------------ A - System Options ----------- #
#    Uncomment the system.power.switch you use
;system.power.switch=ATX_RASPI_R2_6       # http://lowpowerlab.com/
atxraspi/#installation
```

```
;system.power.switch=MAUSBERRY              # http://mausberry-circuits.
myshopify.com/pages/setup
;system.power.switch=REMOTEPIBOARD_2003    # http://www.msldigital.com/
pages/support-for-remotepi-board-2013
;system.power.switch=REMOTEPIBOARD_2005    # http://www.msldigital.com/
pages/support-for-remotepi-board-plus-2015
;system.power.switch=WITTYPI               # http://www.uugear.com/witty-
pi-realtime-clock-power-management-for-raspberry-pi
;system.power.switch=PIN56ONOFF            # https://github.com/recalbox/
recalbox-os/wiki/Add-a-start-stop-button-to-your-recalbox-(EN)
;system.power.switch=PIN56PUSH             # https://github.com/recalbox/
recalbox-os/wiki/Add-a-start-stop-button-to-your-recalbox-(EN)
;system.power.switch=PIN356ONOFFRESET      # https://github.com/recalbox/
recalbox-os/wiki/Add-a-start-stop-button-to-your-recalbox-(EN)
system.power.switch=PIN356PUSHRESET      # https://github.com/recalbox/
recalbox-os/wiki/Add-a-start-stop-button-to-your-recalbox-(EN)
```

Plusieurs systèmes de mise en route/arrêt sont supportés par la distribution. Dans notre cas, nous allons activer la gestion des boutons poussoirs (« temporary switch ») en décommentant la dernière ligne.

Le script de gestion des boutons est lancé par le fichier `/etc/init.d/S92switch` qui utilise le fichier `/recalbox/scripts/powerswitch.sh`. Ce dernier définit les fonctions à utiliser en fonction de la méthode choisie dans le fichier `recalbox.conf` : dans notre cas c'est la fonction `pin356_start()` qui sera lancée :

```
pin356_start()
{
 mode=$1
 python /recalbox/scripts/rpi-pin356-power.py -m "$mode" &
    pid=$!
    echo "$pid" > /tmp/rpi-pin356-power.pid
    wait "$pid"
}
pin356_stop()
{
    if [[ -f /tmp/rpi-pin356-power.pid ]]; then
        kill `cat /tmp/rpi-pin356-power.pid`
    fi
}
```

Cette dernière fait appel au script python `/recalbox/scripts/rpi-pin356-power.py`. Par défaut, il utilise les broches 3, 5 et 6 (GPIO 2, 3 et masse) qui sont utilisées par le module Piglow en I2C. Il nous faut donc utiliser de nouvelles broches. On choisira ici 29, 30 et 31, (GPIO5, masse et GPIO 6). On modifie alors le script `rpi-pin356-power.py` comme suit :

```
import RPi.GPIO as GPIO
import time
import os
import thread
import argparse
```

```
parser = argparse.ArgumentParser(description='power manager')
parser.add_argument("-m", help="mode onoff or push", type=str, required=True)
args = parser.parse_args()

mode = args.m

#Unusable when I2C is enable and used by Piglow
#POWERPLUS = 3 #GPIO 3 on Pin 5
#RESETPLUS = 2 #GPIO 2 On Pin 3

POWERPLUS = 5 #GPIO 5 on Pin 29
RESETPLUS = 6 #GPIO 6 On Pin 31

LED = 14
```

On va ici utiliser les GPIO 5 et 6 sur les pins 29 et 31 du GPIO, et 30 pour la masse. Après un redémarrage, tout devrait fonctionner comme désiré.

Note : Nous avons ici modifié des scripts systèmes. Lors d'une mise à jour de Recalbox, ils seront remplacés par une nouvelle version et la modification sera à effectuer à nouveau pour avoir un module Piglow et des boutons fonctionnels.

CONCLUSION

En guise de conclusion, je vous renvoie vers la vidéo YouTube **[10]** qui présente le résultat de l'intégration du module Piglow dans un boîtier Pitendo embarquant un Raspberry Pi 3. Maintenant c'est à vous de jouer. YM

LIENS

[1] https://shop.pimoroni.com/products/piglow

[2] https://fr.pinout.xyz/pinout/piglow

[3] https://github.com/pimoroni/piglow

[4] https://github.com/Boeeerb/PiGlow

[5] https://www.raspberrypi.org/downloads/raspbian/

[6] http://www.recalbox.com/

[7] https://www.thingiverse.com/thing:2192614

[8] https://github.com/recalbox/recalbox-os/wiki/Add-a-start-stop-button-to-your-recalbox-%28EN%29

[9] https://github.com/pimoroni/sn3218/blob/master/library/sn3218.py

[10] https://www.youtube.com/watch?v=xMrP1M-8r0o

SENSE HAT POUR DONNER DE LA COULEUR ET PLUS À VOTRE RASPBERRY PI

Yann Morère



Les modules d'extension pour notre framboise préférée sont de plus en plus nombreux : carte son, carte de puissance pour moteur pas à pas, contrôleur de lumière et d'effets spéciaux, contrôles arcade, cartes relais, écran TFT, etc. Nous allons ici nous intéresser à un module qui embarque une matrice 8x8 de leds RGB ainsi que des capteurs environnementaux : la carte Sense HAT .Voyons comment la mettre en œuvre.



Le GPIO (*General Purpose Input Output*) du Raspberry Pi nous permet de connecter différents matériels et les piloter suivant différentes méthodes par l'intermédiaire des entrées/sorties (Digital IO, PWM) et des bus de données disponibles : SPI, I2C, DPI. Cette ouverture vers le monde réel a permis d'initier de nombreux projets autour de cartes d'extension dont chacune couvre un besoin bien particulier : son, IoT, moteur pas à pas, etc. On retrouve de nombreux produits chez Adafruit **[1]**, Pimoroni **[2]**, mais aussi des modules réalisés par l'intermédiaire de financements participatifs **[3-4]**. Ces cartes qui se connectent au GPIO et de fait, se positionnent au-dessus du Raspberry Pi, sont nommées des « Hat » (chapeaux). On retrouvera une liste importante, mais non exhaustive de modules à l'adresse **[5]**. Tous ces matériels contribuent à faire du Raspberry Pi un couteau suisse de l'informatique embarquée.

Nous allons dans ces lignes nous concentrer sur la carte Sense HAT **[6]** qui embarque une matrice 8x8 de LEDs RGB ainsi que des capteurs environnementaux, une centrale inertielle et un joystick.

Note : La carte Unicorn Hat [7] embarque elle aussi une matrice de 64 LEDs RGB et aurait pu être utilisée. Sa mise en œuvre fera peut-être l'objet d'un prochain article.

Le but ici est de réaliser l'affichage d'informations venant des capteurs environnementaux, mais aussi des animations de « sprites ». Le gros intérêt de ce type de matériels est qu'il est pilotable via des scripts Python et une API dédiée très simple d'accès.

Note : Dans un jeu vidéo, un sprite est un élément graphique qui peut se déplacer sur l'écran. En principe, un sprite est en partie transparent, et il peut être animé par l'intermédiaire de plusieurs images matricielles qui s'affichent les unes après les autres.

Dans la suite, la distribution utilisée sur le Raspberry Pi 3 de test est une Raspbian Lite d'avril 2017 **[8]**.

Note : Des notions de programmation en langage python sont un plus pour suivre facilement l'article.

1. SENSE HAT ET LA MISSION ASTROPI

Le module Sense HAT ne peut pas être dissocié de la mission AstroPi **[9]** (le module fut d'ailleurs nommé ainsi au départ). Deux Raspberry Pi (appelés Astro Pis) ont été envoyés dans la Station Spatiale Internationale (ISS) dans le cadre de la mission (Principia) de l'astronaute britannique de l'ESA Tim Peake. Ils sont tous deux équipés d'un module Sense HAT qui peut mesurer l'environnement à l'intérieur de la station, détecter la façon dont la station se déplace dans l'espace et récupérer le champ magnétique terrestre. Chaque Astro Pi est également équipé d'un type de caméra différent : Izzy (eh oui, ils ont des noms) possède une caméra infrarouge et Ed dispose d'une caméra dans le spectre visible standard. Chaque Raspberry Pi est placé dans une coque adaptée (cf. figure 1).

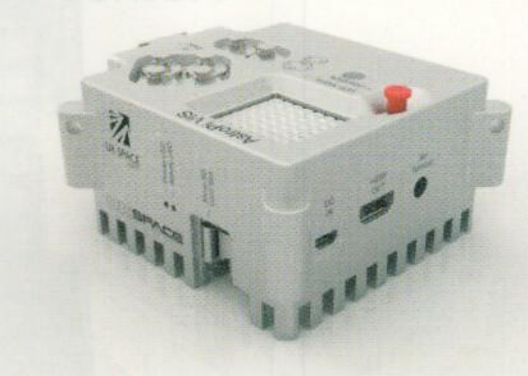


Figure 1

En utilisant les capteurs de la Sense HAT, Tim a exécuté des expériences créées et codées par des étudiants du Royaume-Uni dans la première phase de la mission. Sept équipes ont eu la chance d'envoyer leur code dans l'espace. Leurs expériences se présentent sous la forme de programmes Python écrits et testés par les étudiants (jeux, temps de réaction, détection de personnes, détection de rayonnement grâce aux caméras). Les programmes gagnants ont décollé de la base aérienne de Cap Canaveral en Floride en décembre 2015 et sont arrivés à l'ISS quelques jours avant Tim. Les Astro Pis ont été installés à l'intérieur du module européen Columbus. Les résultats de la première phase sont disponibles à l'adresse **[10]**. Leur mission se poursuit avec la Mission Proxima où des écoliers de toute l'Europe ont rédigé des programmes pour l'astronaute français Thomas Pesquet de l'ESA **[11]**.

2. LE MODULE SENSE HAT EN DÉTAIL

Les fonctionnalités principales de la carte Sense HAT sont mises en valeur dans la figure 2.

On y retrouve les composants suivants **[12]** :

Sur le bus I2C :

- Une centrale de mesure inertielle (*Inertial Measurement Unit* en anglais, souvent appelée IMU) de référence ST LSM9DS1. Ce circuit comporte un accéléromètre 3D, un gyroscope 3D et un magnétomètre 3D combinés dans une seule puce. Il s'agit du composant qui se trouve juste au-dessus du texte « ACCEL/GYRO/MAG ». Le LSM9DS1 peut mesurer des accélérations de ±2g/±4g/±8g/±16g, un champ magnétique de ±4/±8/±12/±16 gauss ainsi qu'une vitesse angulaire de ±245/±500/±2000 degrés/seconde. Il est donc capable de fournir des informations de tangage, roulis et lacet sur les mouvements qu'il subit.
- Un capteur d'humidité relative et capteur de température : référence ST HTS221. Ce capteur fournit le pourcentage d'humidité relative ainsi que la température en degrés centigrades. Il s'agit du composant qui est placé en dessous du texte « HUMIDITY ».

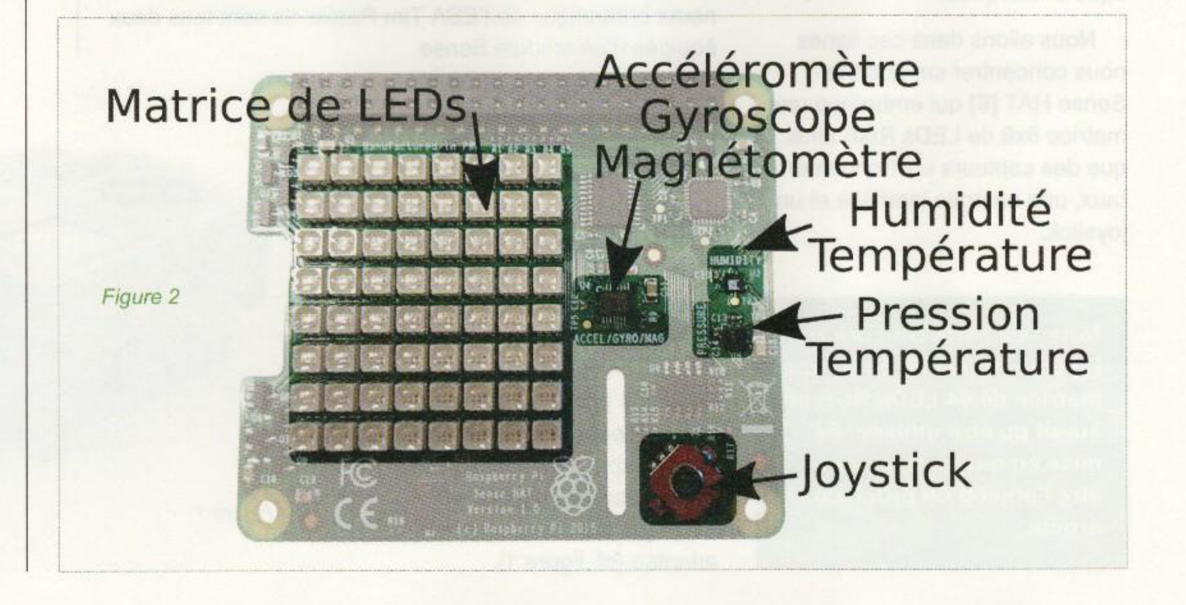


Figure 2

- Un capteur de pression barométrique et capteur de température : référence ST LPS25H (« PRESSURE » sur la carte). Ce circuit peut mesurer une pression absolue comprise entre 260 et 1260 hPa avec une précision d'1Pa. Il intègre une compensation de la température et un convertisseur analogique-numérique 24 bits.

Pilotés par un microcontrôleur Atmel Tiny88 :

- Une matrice 8x8 de 64 LEDs. Sans écran connecté, elle peut être utilisée comme un afficheur. La matrice de LED est pilotée par une combinaison formée d'un pilote de LED à courant constant (un LED2472G) et de l'ATTiny88 Atmel exécutant un firmware qui gère l'affichage 8x8 en RVB avec une résolution de 15 bits.
- Un joystick qui permettra, en cas d'absence de clavier et de souris, de piloter les applications présentes sur le Raspberry Pi. Il dispose de 5 contacts : haut, bas, droite, gauche et clic vertical. Il s'agit du modèle Alps SKRHABE010. Il pourra émuler les touches de direction du clavier et la touche Entrée avec le clic central.

3. INSTALLATION MATÉRIELLE ET LOGICIELLE

L'installation matérielle consiste simplement à enficher la carte sur le connecteur GPIO. Il est possible de mettre en place les entretoises et les vis fournies avec le module afin que ce dernier soit fixé solidement.

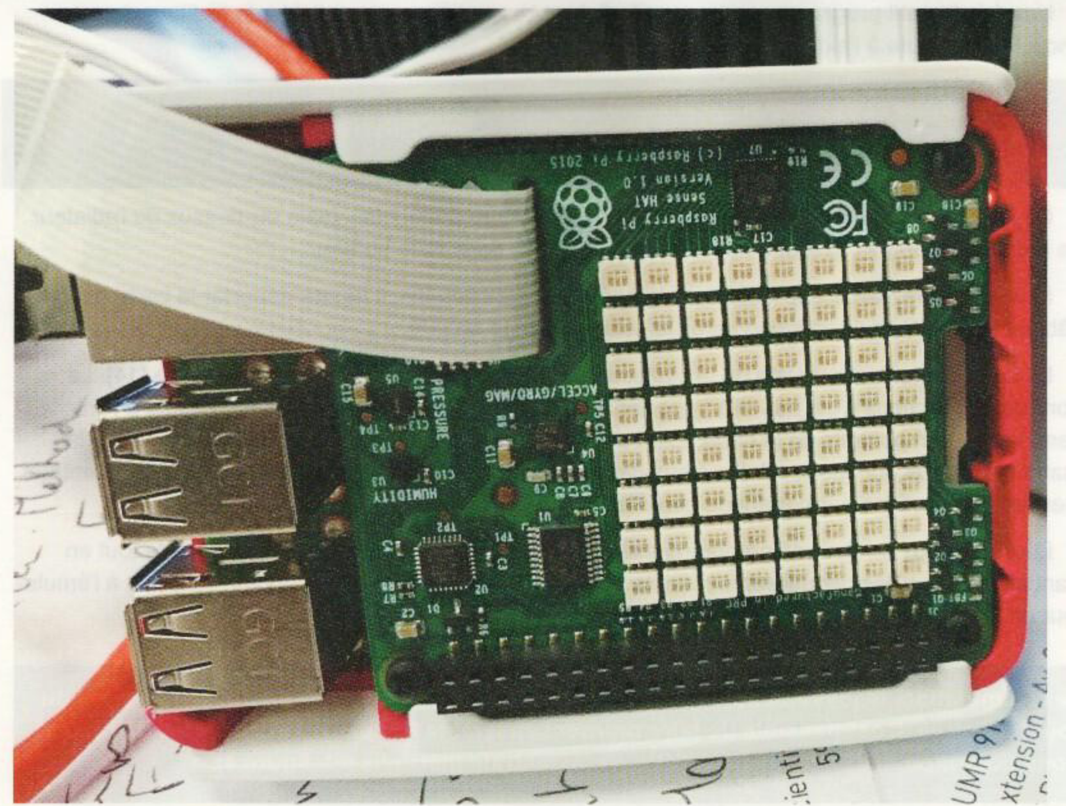


Figure 3

Cette carte ayant vu le jour en 2015, elle est très bien documentée et supportée par la distribution Raspbian. L'installation de ses composants (bibliothèque de gestion de la Sense HAT et de gestion d'image) est très simple :

```
$ sudo apt-get update
$ sudo apt-get install sense-hat
$ sudo apt-get install python-pil
$ sudo reboot
```

Note : Si vous désirez utiliser la version 3 de python pour exécuter vos programmes, il faudra installer les paquets idoines : `python3-sense-hat` et `python3-pil:armhf` disponibles par défaut dans la distribution.

Une fois le Raspberry Pi redémarré, il est possible de tester le bon fonctionnement de la Sense HAT par l'intermédiaire du script `temperature.py` par exemple :

```
from sense_hat import SenseHat
sense = SenseHat()
temp = sense.get_temperature()
print("Temperature: %s C" % temp)
```

Une fois le petit programme saisi dans le fichier `temperature.py` à l'aide de l'éditeur de votre choix, on l'exécute à l'aide de la commande suivante :

```
pi@raspberrypi:~/sensehat $ python temperature.py
Temperature: 34.0726776123 C
pi@raspberrypi:~/sensehat $
```

Ce programme très simple nous affiche la température ambiante... juste au-dessus du radiateur de mon Raspberry Pi 3 ! Donc non, il ne fait pas 34°C dans mon bureau.

Si vous désirez avoir une lecture de température plus précise, il faudra déporter la carte en câblant les GPIO nécessaires d'après le schéma donné en **[13]**.

La documentation complète de la bibliothèque Sense HAT est disponible à l'adresse **[14]**. Le nombre de fonctions n'est pas très important, mais permet d'accéder de manière simple à l'ensemble des composants de la carte. L'API est décomposée en 4 parties : fonctions propres à la matrice de LEDs, fonctions relatives aux capteurs d'environnement, fonctions relatives à la centrale inertielle et fonctions permettant de gérer le joystick.

L'adresse **[15]** contient de nombreux exemples de mises en œuvre des capteurs et le tout en français. Si vous désirez utiliser le module Sense HAT sans l'acheter, c'est possible grâce à l'émulateur en ligne **[16]** (Figure 4).

Note : Lors de la simulation, le joystick de la carte Sense HAT est géré par les touches de direction du clavier et la touche Entrée. Il est possible de faire tourner l'ensemble Raspberry Pi et Sense HAT dans les 3 dimensions pour modifier en simulation l'état de la centrale inertielle (cf. figure 4).

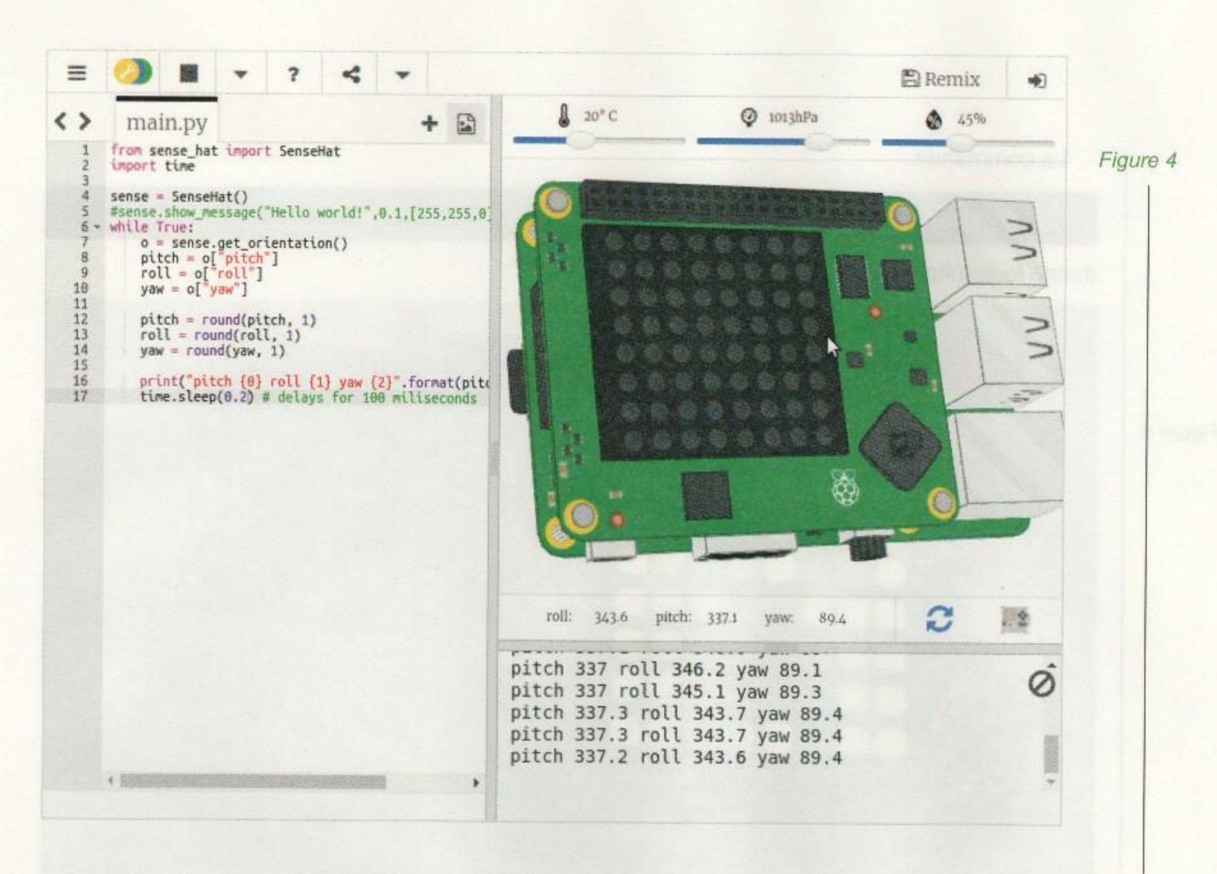


Figure 4

Dans la suite, nous nous focaliserons sur l'utilisation de la matrice de LEDs et les capteurs d'environnement. Si vous désirez mettre en œuvre basiquement la centrale inertielle et le joystick, je vous renvoie vers les pages **[17]** et **[18]**.

4. AFFICHER DES IMAGES

Afin d'afficher une image sur la matrice de LEDs, il faut créer une liste de 64 « sous »-listes qui contiendront la couleur de chaque LED et ensuite fournir cette liste à la fonction `set_pixels`.

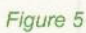
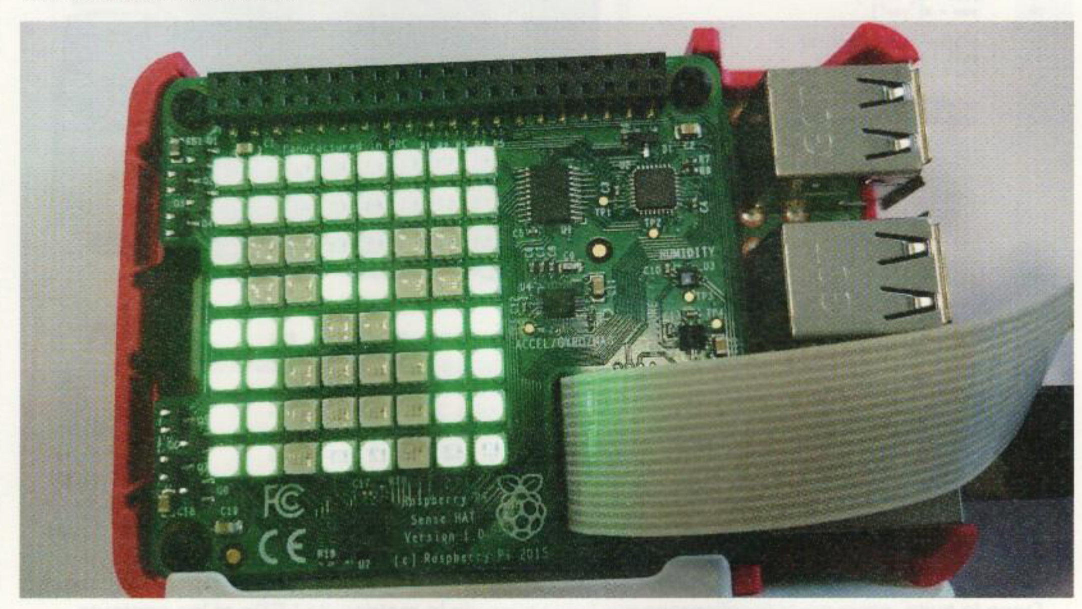
L'exemple python suivant permet d'afficher un « creeper » sur la matrice de LEDs :

```
from sense_hat import SenseHat
sense = SenseHat()
O = (0, 255, 0) # Green
X = (0, 0, 0) # Black
creeper_pixels = [
    O, O, O, O, O, O, O, O,
    O, O, O, O, O, O, O, O,
    O, X, X, O, O, X, X, O,
    O, X, X, O, O, X, X, O,
    O, O, O, X, X, O, O, O,
    O, O, X, X, X, X, O, O,
    O, O, X, X, X, X, O, O,
    O, O, X, O, O, X, O, O
]
sense.set_pixels(creeper_pixels)
```

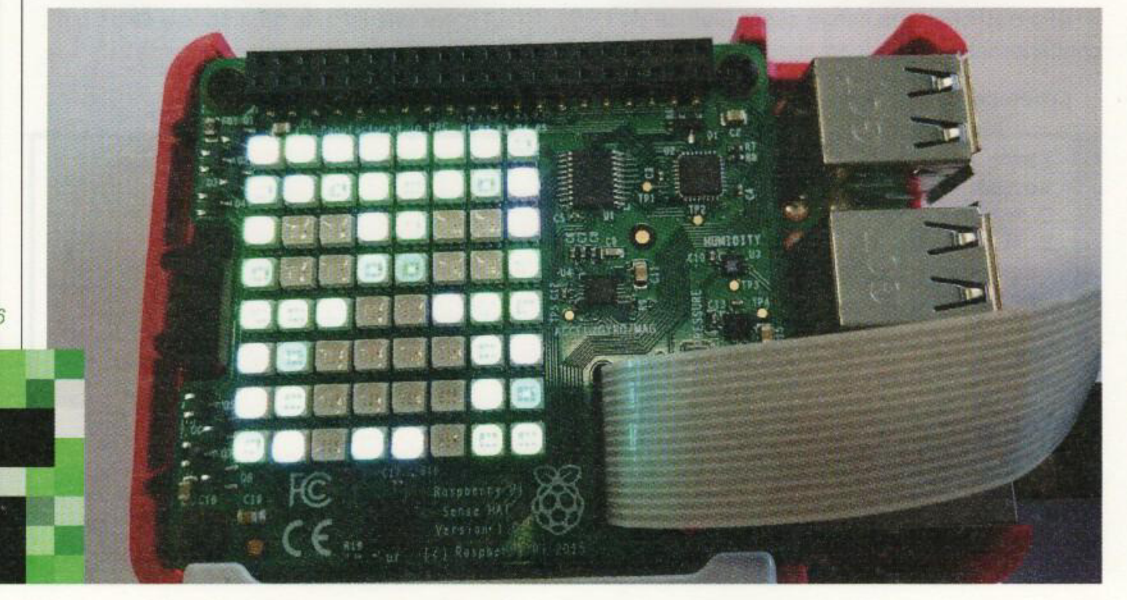
La commande :

```
$ python creeper1.py
```

donne le résultat suivant :



Figure 5

Après avoir instancié un nouvel objet **sensehat**, on définit 2 variables qui vont contenir les couleurs désirées (vert et noir -> éteint). Ensuite, on remplit la liste de 64 pixels avec la couleur désirée pour chaque pixel. Cette liste est linéaire, mais on la représente dans le code sous la forme d'une matrice 8x8 afin de vérifier visuellement la position de nos couleurs. Ensuite, on envoie cette liste à l'affichage.


Figure 6

Cependant, la manière la plus simple d'afficher des images est d'utiliser la fonction load_images disponible dans l'API. Cette dernière permet d'afficher une image PNG de 8x8 pixels sur la matrice de LEDs. Elle renvoie aussi la liste de 64 pixels qui compose l'image.

On pourra par exemple afficher un « creeper » (avec toutes ses nuances de couleurs) sans être obligé de déclarer les variables associées (figure 6).

La fonction load_images possède une option intéressante, qui permet de charger l'image, mais de ne pas l'afficher immédiatement sur la matrice de LEDs. Si l'on récupère la liste des pixels dans une variable, on pourra alors afficher notre image ultérieurement. Le programme suivant montre l'utilisation de la fonction de deux manières différentes : affichage immédiat et affichage différé. La méthode time.sleep() permet de temporiser l'exécution du programme. La méthode sense.clear() permet d'éteindre tous les pixels de la matrice.

```
from sense_hat import SenseHat
import time

sense = SenseHat()
sense.clear()
sense.load_image("sprites/creeper.png",True)
time.sleep(1)
sense.clear()
time.sleep(1)
pixel_list = sense.load_image("sprites/creeper.png",False)
print("Wait!!!")
time.sleep(1)
print("Display!!!")
sense.set_pixels(pixel_list)
time.sleep(1)
sense.clear()
```

Cette méthode d'affichage différé sera très utile dans le cas de la création de petites animations. On chargera dans un premier temps tous les « sprites » de notre animation en mémoire, et ensuite on les affichera en temps voulu.

5. RÉALISER UNE ANIMATION

5.1 Animer simplement un texte

L'animation la plus simple à réaliser sur la Sense HAT est le défilement horizontal de texte. Ceci est réalisé par l'intermédiaire de la méthode show_message qui ne retourne rien, mais qui prend en paramètres la chaîne de caractères à afficher, la vitesse de défilement, la couleur des lettres et la couleur de l'arrière-plan.

```
from sense_hat import SenseHat
sense = SenseHat()
sense.show_message(text_string="Sir... On en a gros!!!", scroll_speed=0.1,
text_colour=[255,0,0], back_colour=[0,0,255] )
sense.clear()
```

Le script ci-dessus fait défiler à la vitesse d'une colonne toutes les 0,1s, le texte en rouge sur fond bleu.

5.2 Animer des sprites

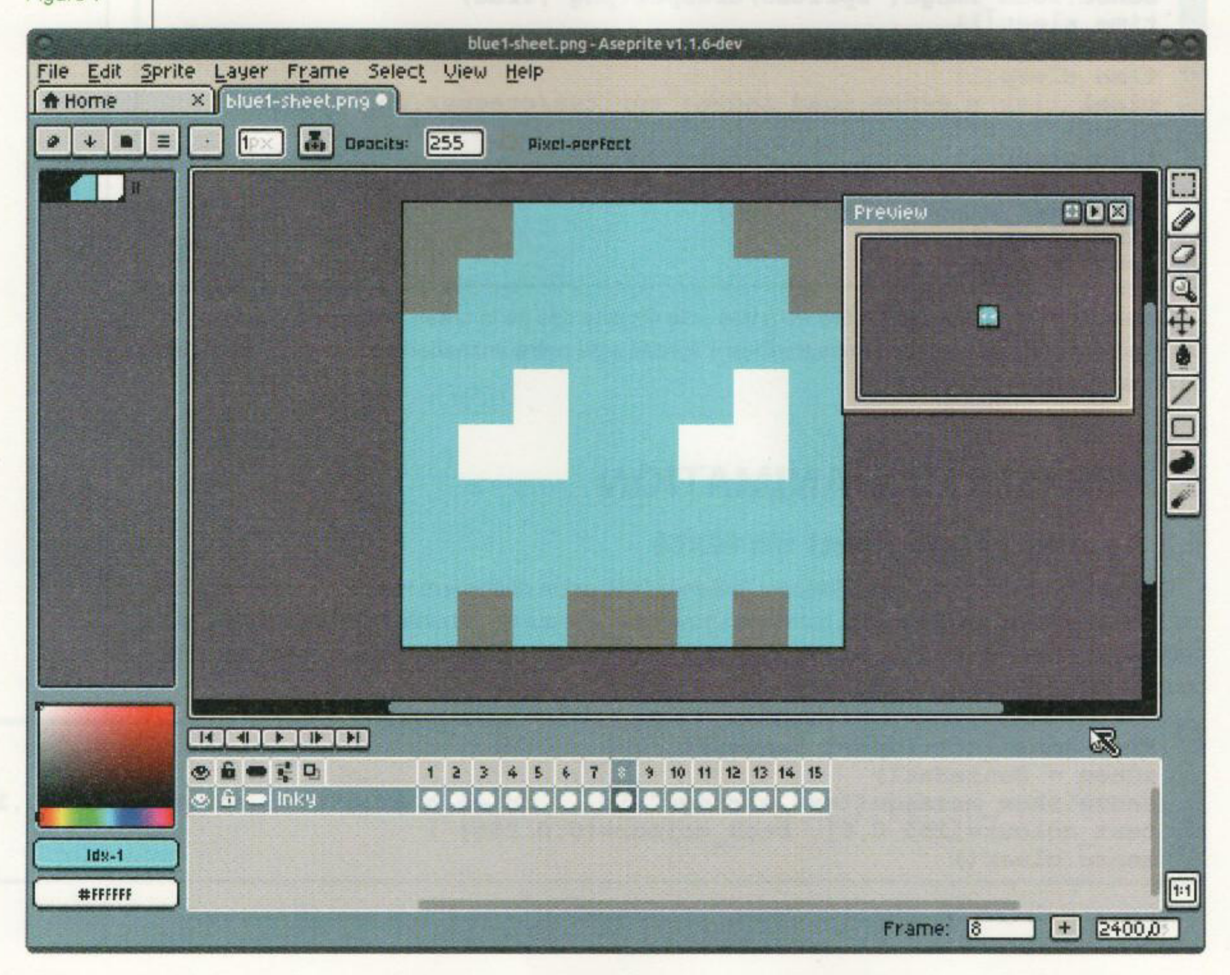
Si maintenant vous désirez animer des « sprites » à la place d'un texte, il va falloir créer chaque image de votre animation et les enchaîner.

Pour cela, on peut utiliser le logiciel aseprite **[19]** qui est disponible sur les distributions Linux classiques comme Debian, Ubuntu, etc. Il ne s'agit pas d'un logiciel libre, mais ses sources sont disponibles à l'adresse **[20]**.

Aseprite est dédié à la création de « sprites » animés. Il permet la sauvegarde aux formats PNG/GIF, il gère plusieurs modes de couleurs (indexé, RGBA, niveaux de gris), utilise les notions de calques (*layers*) et cadres (*frames*). Son interface nous ramène un peu au bon vieux temps de l'Amiga et de l'Atari, ce qui n'est pas pour me déplaire :) (figure 7). Sa prise en main est assez simple et une très bonne documentation est disponible à l'adresse **[21]**.

Dans l'animation que nous allons créer, nous allons faire se déplacer Inky (le petit fantôme bleu de Pacman) sur la matrice de LEDs. Pour cela, on crée un nouveau « sprite » de 8x8 pixels et l'on dessine notre petit fantôme comme décrit en figure 7. On fera attention de choisir un fond transparent (cela correspondra aux LEDs éteintes).

Figure 7



L'animation va se dérouler en 15 images. Il faudra donc créer 15 « frames » et chacune d'elles sera une partie du déplacement de notre fantôme (cf. figure 7). On crée une nouvelle « frame » via le menu ***Frame > New Frame***. Pour aller plus vite, et ne pas redessiner à chaque fois tous les pixels, on peut sélectionner une zone rectangulaire et la déplacer. On peut ensuite enregistrer chaque « frame » dans un fichier PNG en une seule action en utilisant le menu ***Save as***. Pour notre exemple, cela génèrera les fichiers `inky1.png`, `inky2.png`... `inky15.png` (figure 8).

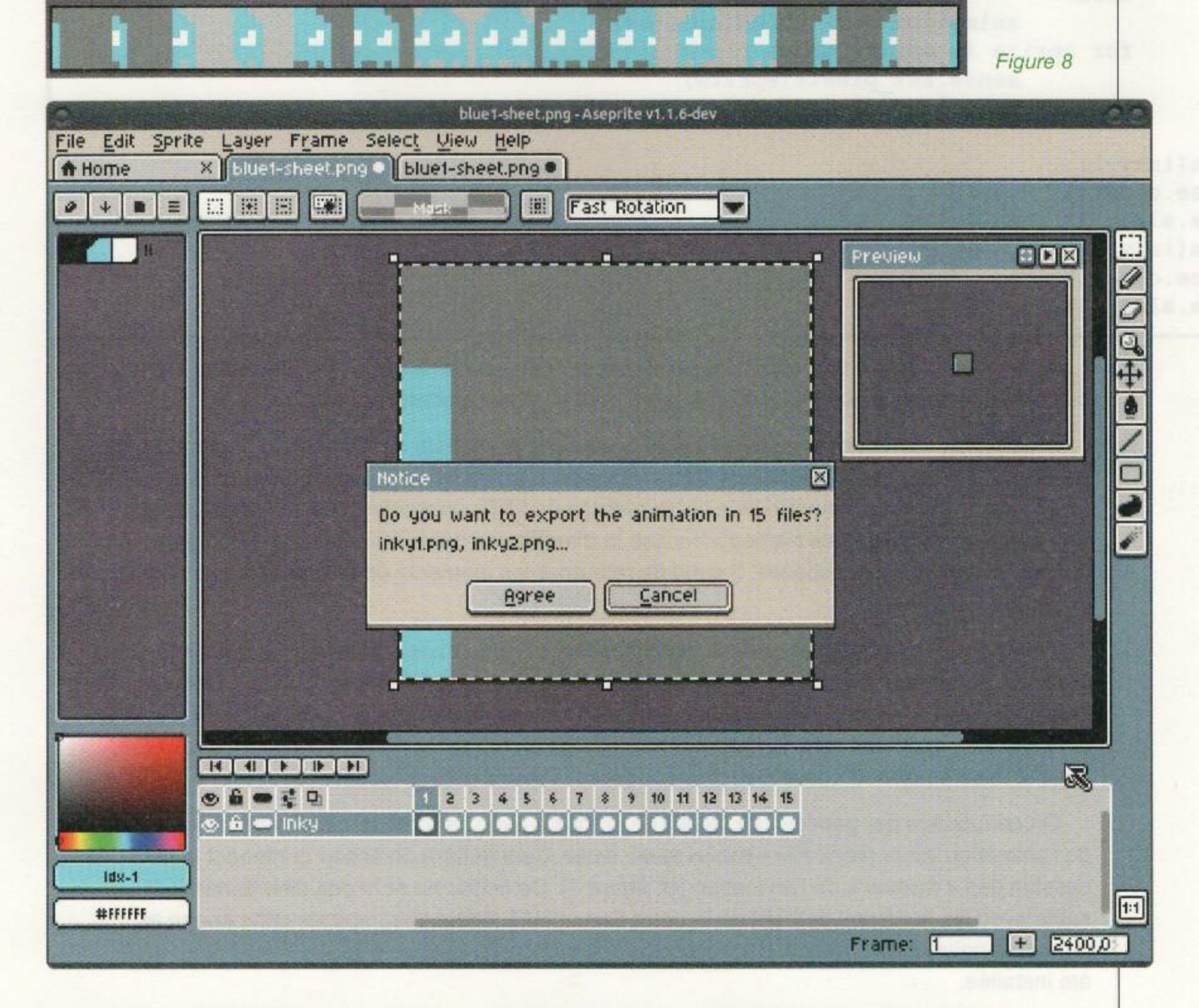


Figure 8

Après avoir téléversé les « sprites » sur le Raspberry Pi, on écrit le petit programme `inky_anim.py` qui va réaliser notre animation :

```
from sense_hat import SenseHat
import time

sense = SenseHat()
sense.set_rotation(180)
```

```
inky=[]
for nbr in range(1, 15):
        inky.append(sense.load_image("sprites/inky"+str(nbr)+".png", redraw=False))

def Move(sprites, direction):
        if direction:
                animation = sprites
        else:
                animation = reversed(sprites)
        for sprite in animation:
                sense.set_pixels(sprite)
                time.sleep(0.1) # delays for 100 miliseconds

Move(inky,1)
sense.clear()
time.sleep(1)
Move(inky,0)
sense.clear()
time.sleep(1)
```

Après avoir déclaré une liste vide, on va ajouter dans cette dernière les différentes « frames » de notre animation grâce à la méthode **sense.load_image** et une boucle.

Ensuite, on définit une fonction qui aura pour objet de réaliser l'animation du déplacement de notre « sprite » sur la matrice de LEDs. Pour cela, on va afficher successivement les différentes « frames » de notre liste sur la matrice de LEDs. Notre fonction prend deux paramètres qui sont la liste des frames ainsi que la direction de déplacement. Ainsi pour effectuer le déplacement en sens opposé, il suffit de retourner les éléments de la liste et de les afficher (méthode **reversed**).

Plutôt que de charger des images distinctes une par une dans une liste, il est possible, avec aseprite, de générer une seule image (feuille de « sprite »/« sprite sheet) », qui contiendra l'ensemble des « frames ». Il nous faut alors extraire de cette image globale chaque « frame » pour l'insérer dans une liste. Le gain est intéressant en terme de nombre de fichiers générés, mais la mise en œuvre est un peu plus complexe. Voyons cela en détail.

On commence par générer la « feuille de sprites » avec aseprite en effectuant un export de l'animation via le menu ***File > Export Sprite Sheet***. Cela génère un fichier contenant la succession des « frames » de l'animation (cf. figure 7). Ce fichier ne sera pas directement utilisable avec les fonctions de l'API de la carte Sense HAT. Il faut décomposer cette image et pour cela nous aurons besoin de la bibliothèque python de gestion d'image Pillow qui a déjà été installée.

Le morceau de code suivant permet de tester la présence de la bibliothèque PIL.

```
try:
    from PIL import Image
except ImportError:
    exit("This script requires the pillow module\nInstall with: sudo pip
install pillow")
```

La fonction suivante, lit une image PNG contenant une série de « frames » de dimension 8x8 pixels et stocke les images dans une liste :

```
def readSprites(filename):
      img = Image.open(filename)
      sprites_list = []
      for o_x in range(int(img.size[0]/8)):
          for o_y in range(int(img.size[1]/8)):
              sprite=[]
              for x in range(8):
                  for y in range(8):
                      pixel = img.getpixel(((o_x*8)+y,(o_y*8)+x))
                      r, g, b = int(pixel[0]),int(pixel[1]),int(pixel[2])
                      sprite.append([r,g,b])
              sprites_list.append(sprite)
      return sprites_list
```

Je ne rentrerai pas dans le détail des explications de la fonction, mais on peut tout de même remarquer que les pixels de chaque image sont stockés un par un. Ensuite, on réalise la fonction qui va animer le « sprite » :

```
def animateSprite(sprite,delay,direction):
#     print(len(sprite), delay, direction)
      if (len(sprite)!=0):
              for index,spr in enumerate(sprite):
                      sense.clear()
                      if direction=='b':
                              flipped = []
                              for i in range(8):
                                      offset = i * 8
                                      flipped.extend(reversed(spr[offset:offset + 8]))
                              sense.set_pixels(flipped)
                      else:
                              sense.set_pixels(spr)
                      time.sleep(delay)
      else:
              time.sleep(delay)
      sense.clear()
```

Cette dernière prend en paramètres, la liste de « frames » qui compose le « sprite », la vitesse de défilement (réglée en fait par une temporisation entre deux affichages) et la direction de déplacement. On remarquera la facilité avec laquelle il est possible de retourner l'image grâce à l'utilisation de la méthode **reversed**.

Note : Il aurait été aussi possible de réaliser le retournement horizontal de l'image grâce à l'utilisation de la méthode flip_h de l'API de la Sense HAT.

Voici le programme principal qui permet de gérer de mini scenarii d'animation :

```
none=[]
skulls = readSprites('sprites/skulls.png')
clyde = readSprites('sprites/clyde-sheet.png')
blinky = readSprites('sprites/blinky-sheet.png')
inky = readSprites('sprites/inky-sheet.png')
deadghost = readSprites('sprites/deadghost-sheet.png')
pinky = readSprites('sprites/pinky-sheet.png')
pacman = readSprites('sprites/pacman-sheet.png')

animation_order = [[pacman, 0.12, 'f'], [none,0.8,'f'] , [clyde,0.1,'f'],
[none,0.8,'f'], [deadghost, 0.03, 'b'], [none,0.5,'f'], [pacman, 0.03, 'b']]

for i in range (len(animation_order)):
        [animation,delay,direction] = animation_order.pop(0)
        animateSprite(animation,delay,direction)
        animation_order.append(animation)
```

Dans un premier temps, on charge tous les « sprites » en mémoire. Le « sprite » **none**, qui ne contient pas d'image, est présent pour réaliser une temporisation entre les animations. On crée un scénario d'animation en enchaînant une liste de listes de paramètres qui seront passés à la fonction **animateSprite**. Dans cet exemple, Pacman est poursuivi par Clyde dans un premier temps, mais ensuite c'est Clyde qui fuit devant Pacman. Le script complet **demo_animation.py** est disponible à l'adresse **[22]**.

5.3 Ajouter des polices au texte qui défile

Par défaut, la méthode **show_message** de la bibliothèque permet de faire défiler du texte, mais ses options ne permettent pas de choisir la police de caractères. Dans cette partie, nous allons utiliser figlet **[23]** afin de générer une « pseudo police » de caractères pour modifier l'apparence du texte à faire défiler.

« FIGlet est un logiciel qui crée des bannières textuelles dans différentes polices d'écriture. Chaque caractère est composé d'un amas de plus petites lettres à la manière de l'art ASCII. » **[24]**

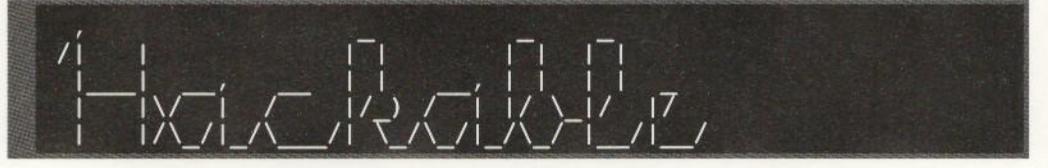
On commence par installer **figlet** et le module python éponyme :

```
$ sudo apt-get install figlet python-pyfiglet
```

On peut alors l'utiliser en ligne de commandes comme suit :

```
$ figlet -d figlet_fonts/ -f script "Hackable"
```

Ce qui donne le résultat suivant après avoir téléchargé le fichier **script.flf** dans le répertoire **figlet_fonts** depuis l'adresse **[25]** :



Note : Certaines polices de caractères sont déjà présentes dans le répertoire /usr/share/figlet/.

Cependant pour notre application, nous allons privilégier les polices figlet qui possèdent un nombre réduit de caractères pour composer les lettres afin de les remplacer par des couleurs différentes :

```
$ figlet -d figlet_fonts/ -f marquee "Hackable"
.::     .::                 .::              .::       .::
.::     .::                 .::              .::       .::
.::     .::   .::      .::::.::  .::   .::   .::       .::   .::
.:::::: .:: .::   .::  .::    .:: .::  .::  .:: .:: .::   .:: .:   .::
.::     .::.::    .:: .::     .:.::  .::    .:: .::  .:: .::.:::::: .::
.::     .::.::    .:: .::     .:: .:: .::   .:: .::  .:: .::.:
.::     .:: .:: .:::      .:::.::  .::   .:: .:::.:: .:: .:::  .::::
```

Les polices suivantes, en choisissant convenablement les couleurs, donnent des résultats satisfaisants : *banner4*, *3x5*, *3-d*, *alligator*, *alligator2*, *contrast*, *marquee*, *o8*, *sblood*, *starwars*, *stellar* et *univers*.

Cependant, avec l'utilisation de **figlet**, il n'est plus possible de mettre en œuvre la méthode **show_message()** de la bibliothèque. Il va nous falloir réécrire cette fonction.

On commence par générer notre texte **figlet** à l'aide la fonction adéquate :

```
try:
    from pyfiglet import figlet_format
except ImportError:
    exit("This script requires the pyfiglet module\nInstall with: sudo pip
install pyfiglet")


TXT = "Hackable is fun!!!"
figletText = figlet_format(TXT+' ', "3-d", width=1000)
print(figletText)
```

Ceci nous donne le résultat en figure 9.

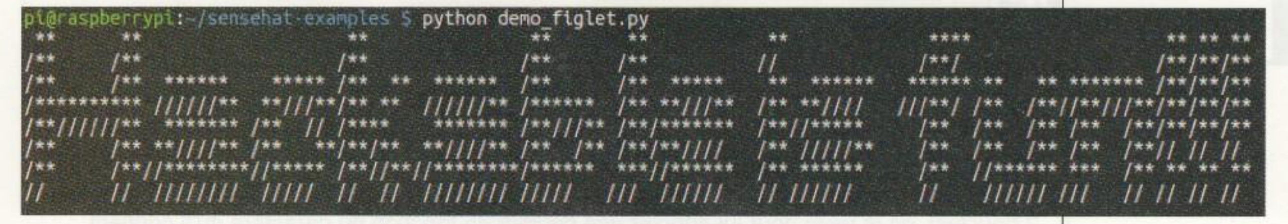


Figure 9

On écrit ensuite la fonction qui va afficher ce texte à l'aide d'une matrice glissante de 8x8 pixels. Cette fonction remplace aussi les caractères par des couleurs : ici les « / » seront affichés en blanc et les « * » en rouge.

```
textMatrix = figletText.split("\n")[:width] # width should be 8
textHeight = len(textMatrix)
if textMatrix[textHeight-1]=='':
        textHeight = textHeight -1
textWidth = len(textMatrix[0]) # the total length of the result from figlet
width = 8
height = 8
i=-1
def step():
    global i
    if i>=100*textWidth : # avoid overflow when looping forever
        i = 0
    else:
        i=i+1
    for h in range(height):
        for w in range(textHeight):
            hPos = (i+h) % textWidth
            chr = textMatrix[w][hPos]
            if chr in ['/', '\\', '_', '(',')','d','8','P', 'b', 'Y', '8',
'o', '^', '|', '-', '`', 'a', '"' ]:
                sense.set_pixel(width-w-1, h, 255, 255, 255)
            elif chr == '#':
                sense.set_pixel(width-w-1, h, 255, 0, 0)
            elif chr == '*':
                sense.set_pixel(width-w-1, h, 255, 0, 0)
            elif chr == ':':
                sense.set_pixel(width-w-1, h, 255, 255, 255)
            elif chr == '.':
                sense.set_pixel(width-w-1, h, 128, 255, 128)
            elif chr == '+':
                sense.set_pixel(width-w-1, h, 255, 0, 255)
            elif chr == '@':
                sense.set_pixel(width-w-1, h, 255, 0, 0 )
            elif chr == '!':
                sense.set_pixel(width-w-1, h, 255, 128, 128)
            elif chr == '\'':
                sense.set_pixel(width-w-1, h, 255, 0, 128)
            else:
                sense.set_pixel(width-w-1, h, 0, 0, 0)
```

Dans un premier temps, on génère les variables nécessaires : longueur et hauteur du texte en pixels afin de connaître le nombre d'itérations. Lorsque l'on a réalisé 100 itérations d'affichage du texte complet, on réinitialise le compteur afin d'éviter un potentiel « overflow ». Une fois chaque pixel sélectionné dans la matrice (par l'intermédiaire de l'enchaînement des 2 boucles), on le teste pour lui affecter une couleur en fonction du caractère qu'il contient. Finalement, on l'affiche.

Le programme complet **demo_figlet.py** est disponible à l'adresse **[22]** et une petite vidéo de l'animation est disponible à l'adresse **[26]**.

6. HORLOGE ET PLUS...

Nous allons maintenant regrouper l'affichage de texte et d'animation à l'intérieur d'un petit programme qui va transformer notre Raspberry Pi et sa Sense HAT en un afficheur à LEDs sur lequel on diffusera, la date, l'heure, la température, l'hygrométrie et la pression atmosphérique. Chaque affichage textuel sera précédé d'une petite animation graphique.

Comme notre Raspberry Pi ne possède pas de module RTC (*Real Time Clock*), nous allons le synchroniser sur un serveur de temps au lancement du programme. Pour cela, nous aurons besoin du programme **ntpdate** ainsi que d'un serveur de temps. Afin d'avoir la date en français, il convient d'ajouter la locale « fr_FR.UTF8 » à l'aide de la commande :

```
$ sudo dpkg-reconfigure locales
```

Ensuite, on fixe la langue choisie pour l'affichage des informations relatives au temps dans notre script python à l'aide du code suivant :

```
# Let's set a non-US locale
import locale
locale.setlocale(locale.LC_TIME, "fr_FR.UTF8")
```

Puis on le met à l'heure à l'aide du code suivant qui est en fait une succession de commandes Bash :

```
print("Setting time")
bashCommand = "sudo service ntp stop; sudo ntpdate 193.50.119.254; sudo
service ntp start"
os.system(bashCommand)
now = datetime.datetime.now()
print(now.strftime("Date : %d %B %Y Heure : %H:%M"))
```

On commence par stopper le service NTP, afin de pouvoir lancer la commande de mise à l'heure réalisée par le programme **ntpdate**. Il faudra bien sûr adapter l'adresse IP à celle de votre serveur de temps. À l'exécution, on obtient le retour suivant :

```
Setting time
12 Jun 17:13:23 ntpdate[17902]: adjust time server 193.50.119.254 offset
0.000010 sec
Date : 12 juin 2017 Heure : 17:13
```

Après avoir chargé tous nos « sprites » à l'aide du code suivant (exemple pour l'animation de inky) :

```
inky=[]
for nbr in range(1, 15):
        inky.append(sense.load_image("sprites/inky"+str(nbr)+".png", redraw=
False))
```

On peut écrire la boucle infinie qui va enchaîner l'affichage des informations et des animations :

```
while (True):
        now = datetime.datetime.now()
        pressure = sense.get_pressure()
        temp = sense.get_temperature()
        humidity = sense.get_humidity()
        Move(inky,0)
        sense.clear()
        sense.show_message(now.strftime("Date : %d %B %Y "), text_colour=[0, 255, 255])
        Move(blinky,0)
        sense.clear()
        sense.show_message(now.strftime("Heure : %H:%M"), text_colour=[255, 0, 0])
        Move(clyde,0)
        sense.clear()
        sense.show_message("Temperature : "+"{0:.1f}".format(temp)+" C", text_
colour=[255, 102, 0])
        Move(pinky,0)
        sense.clear()
        sense.show_message("Pression : "+"{0:.0f}".format(pressure)+" mB", text_
colour=[255, 0, 255])
        Move(pacmanb,1)
        sense.clear()
        sense.show_message("Hygrometrie : "+"{0:.0f}".format(humidity)+"%", text_
colour=[255, 255, 0])
        Move(deadghost,0)
        sense.clear()
        Move(pacmanb,1)
        sense.clear()
        time.sleep(0.5)
```

Ce code source est très simple. Le seul point important concerne la mise en forme des valeurs de température, pression et hygrométrie afin de limiter l'affichage des chiffres décimaux.

Le programme complet **dmdclock.py**, ainsi que les « sprites » sont disponibles à l'adresse **[22]**.

Finalement, on peut lancer ce petit programme au démarrage de notre Raspberry Pi en ajoutant la ligne suivante au fichier **/etc/rc.local** avant le « **exit 0** » final.

```
/home/pi/sensehat-examples/dmdclock.py &
```

On fera attention de mettre le chemin absolu vers les « sprites » lors de leur lecture et stockage en mémoire. Le cas échéant le programme python ne les trouvera pas et s'arrêtera. L'adresse **[27]** vous propose une vidéo du programme final en fonctionnement.

CONCLUSION

En guise de conclusion, je vous renvoie vers les pages **[28]** et **[29]** qui vous proposent de programmer deux petits jeux qui utilisent la centrale inertielle et le joystick. Maintenant c'est à vous. YM

RÉFÉRENCES

[1] https://www.adafruit.com/category/286

[2] https://shop.pimoroni.com/collections/hats

[3] https://www.indiegogo.com/projects/pisound-audio-midi-interface-for-raspberry-pi#/

[4] https://www.kickstarter.com/projects/364371217/caphat-a-raspberry-pi-capacitive-keypad-hat-access

[5] https://pinout.xyz/boards#

[6] https://www.raspberrypi.org/products/sense-hat/

[7] https://shop.pimoroni.com/products/unicorn-hat

[8] https://www.raspberrypi.org/downloads/raspbian/

[9] https://astro-pi.org/

[10] https://astro-pi.org/principia/science-results/

[11] https://astro-pi.org/proxima/

[12] http://www.framboise314.fr/sense-hat-un-tour-dans-les-etoiles

[13] https://pinout.xyz/pinout/sense_hat

[14] https://pythonhosted.org/sense-hat/api/

[15] http://wiki.mchobby.be/index.php?title=RASP-SENSE-HAT-ASTRO-PI

[16] https://www.raspberrypi.org/learning/astro-pi-guide/emulation.md

[17] https://www.raspberrypi.org/learning/astro-pi-guide/sensors/movement.md

[18] https://www.raspberrypi.org/learning/astro-pi-guide/inputs-outputs/joystick.md

[19] https://www.aseprite.org/

[20] https://github.com/aseprite/aseprite/

[21] https://www.aseprite.org/docs

[22] https://github.com/ian57/sensehat-examples

[23] http://www.figlet.org/

[24] https://fr.wikipedia.org/wiki/FIGlet

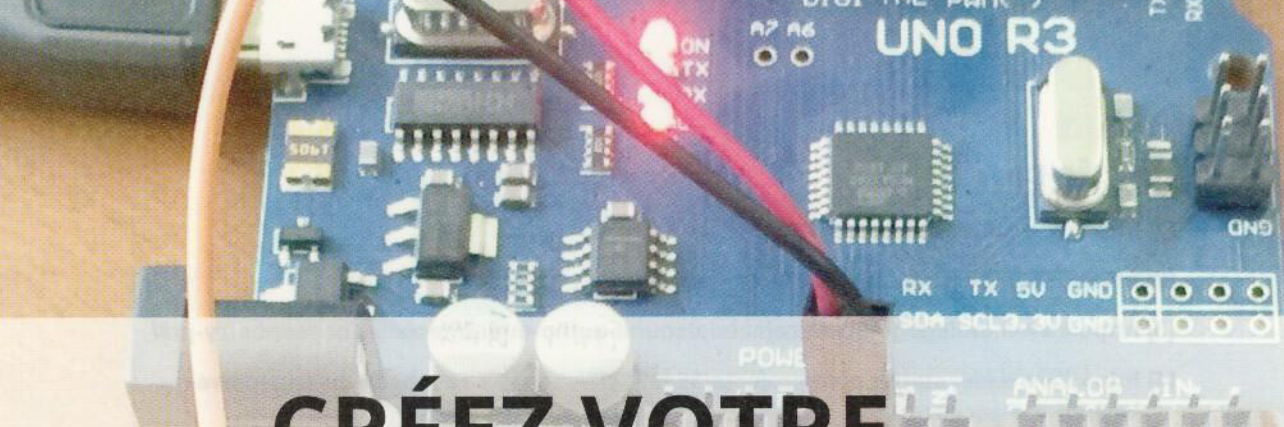
[25] http://www.figlet.org/fontdb.cgi

[26] https://youtu.be/61VepIPmyLQ

[27] https://youtu.be/SOPMpa3ziWk

[28] https://www.raspberrypi.org/learning/sense-hat-pong/

[29] https://www.raspberrypi.org/learning/sense-hat-marble-maze/



CRÉEZ VOTRE ORDINATEUR 8 BITS SUR PLATINE À ESSAIS : LE PROCESSEUR

Denis Bodor

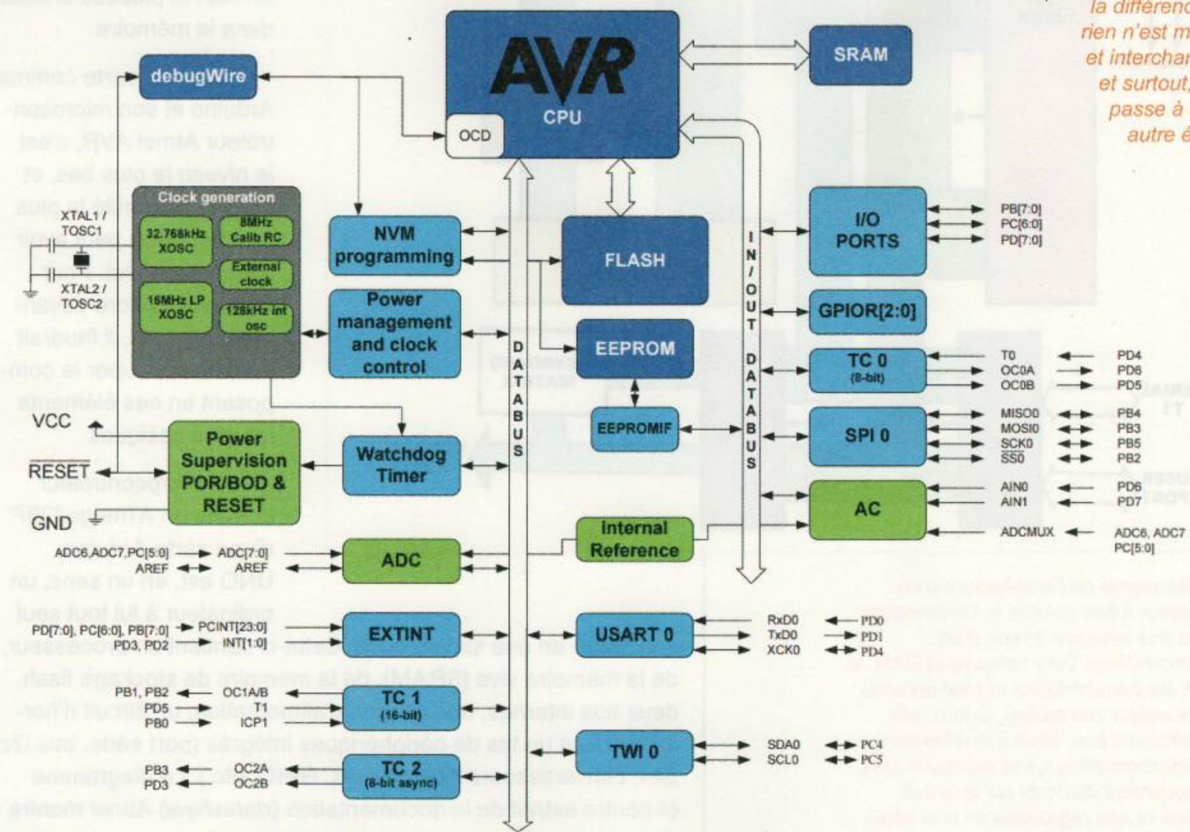


Un PC, un Mac, une carte Raspberry Pi, un Arduino... tout ceci paraît tellement simple de nos jours. Quelques clics, quelques commandes ou quelques lignes de code et voilà la chose s'anime gaiement selon notre bon plaisir. En réalité, cette simplicité n'existe pas et n'est que le résultat d'une dissimulation de la complexité qui, elle, a toujours été présente dans ce domaine, depuis ces toutes premières heures et n'a jamais disparue. Renouer avec cette savoureuse complexité, cette technicité historique, est justement le but de cet article et d'autres qui suivront à l'avenir.

Pourquoi ?! Pourquoi donc vouloir aujourd'hui, alors que tout est maintenant rendu très accessible, chercher à renouer avec quelque chose qui était indispensable il y a des dizaines d'années ? La première réponse qui me viendrait à l'esprit est tout simplement « parce que c'est amusant ». Mais ceci n'explique finalement pas grand-chose et est avant tout une affaire de goût et de perception. La raison pour laquelle une telle approche est amusante ou plaisante est l'envie de satisfaire une certaine curiosité, d'accéder à l'essence des choses, et surtout d'apprendre toujours plus. Sans oublier, bien sûr, la dose de dopamine qui accompagne invariablement une expérimentation réussie. Comme le disait Corneille, à programmer sans difficulté, on compile sans plaisir (propos originaux qui ont étrangement été détournés ensuite pour devenir intelligibles par tous, si, si).

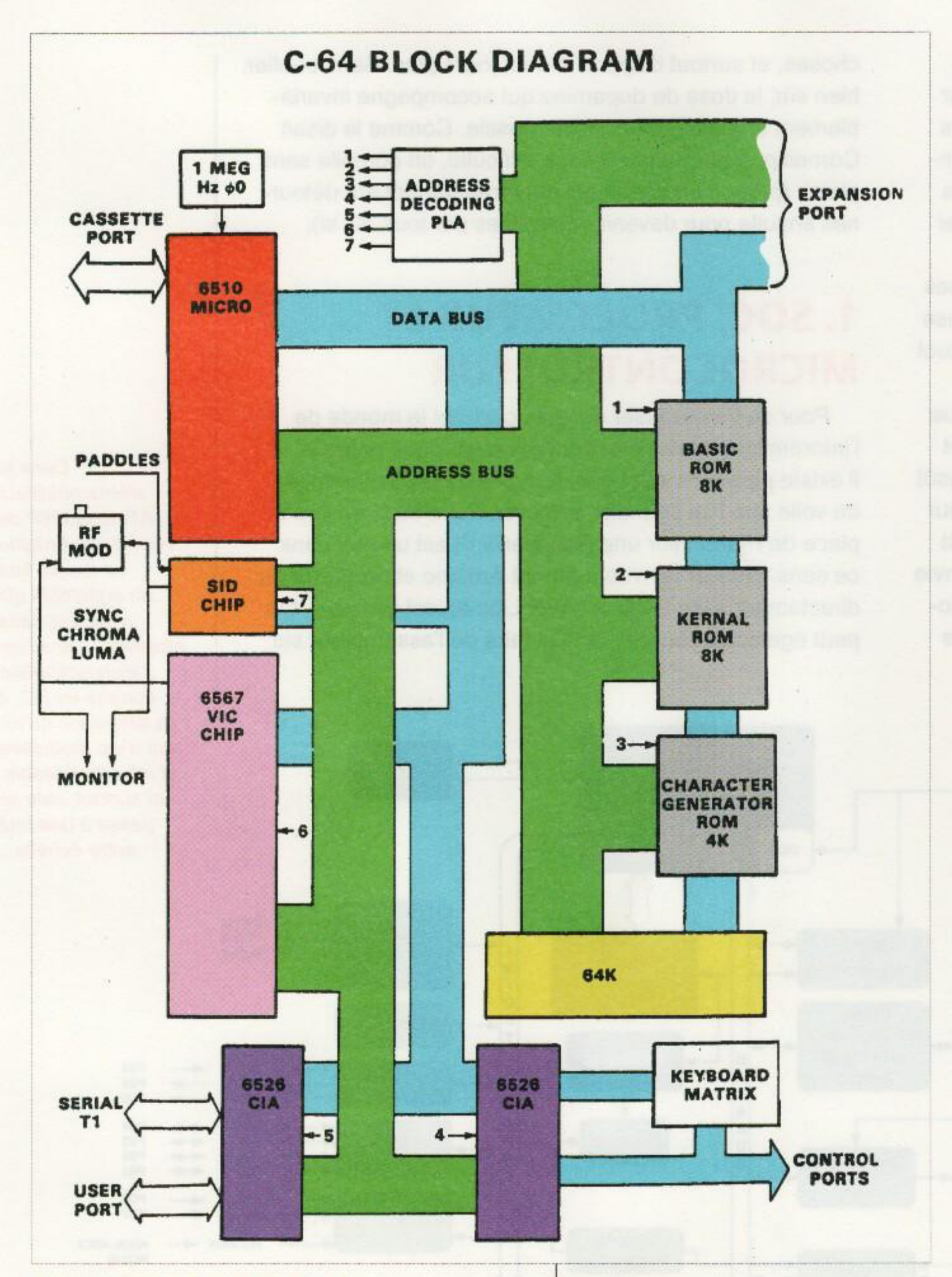
1. SOC, PROCESSEUR ET MICROCONTRÔLEUR

Pour se débarrasser du voile cachant le monde de l'informatique et de l'électronique numérique actuelle, il existe plusieurs solutions, tout dépend de la quantité de voile que l'on souhaite soulever. Faire du C en lieu et place de Python sur une Raspberry Pi est un pas dans ce sens. Oublier l'environnement Arduino et programmer directement avec GCC et l'AVR Libc en est un autre. On peut également écarter le C et faire de l'assembleur sur



Dans le microcontrôleur ATmega328P de votre carte Arduino se trouve tout un ensemble qui n'est pas sans rappeler la structure d'un ordinateur comme un PC, à la différence qu'ici, rien n'est modulaire et interchangeable, et surtout, cela se passe à une tout autre échelle...





Le diagramme de l'architecture d'un ordinateur 8 bits comme le Commodore 64 est très similaire à celle d'un microcontrôleur. On y retrouve la RAM, la ROM, les périphériques et bien entendu le processeur (en rouge), le tout relié par différents bus. Mais à la différence d'un microcontrôleur, ces éléments sont physiquement distincts sur le circuit imprimé et non regroupés en une seule puce.

microcontrôleur Atmel AVR. Le langage C qui, du temps où j'étais en classe, était considéré comme un langage de haut niveau alors qu'il est maintenant dit de bas niveau (plus le niveau est bas, plus on est proche du matériel). Si nous poussons plus loin encore, oublions également l'assembleur, qui est traditionnellement produit par le compilateur C (ou écrit à la main) et écrivons du code machine, les instructions binaires directement utilisées par le processeur, traduites à la main et placées ensuite dans la mémoire.

Avec une carte comme Arduino et son microcontrôleur Atmel AVR, c'est le niveau le plus bas, et donc la proximité la plus grande, qu'on peut avoir avec le matériel. Pour désosser encore davantage tout cela, il faudrait pouvoir découper le composant en ces éléments les plus basiques.

Un microcontrôleur comme un ATmega328P d'une carte Arduino UNO est, en un sens, un ordinateur à lui tout seul concentré en une simple puce. Celui-ci contient un processeur, de la mémoire vive (SRAM), de la mémoire de stockage flash, deux bus internes, une gestion d'alimentation, un circuit d'horloge et tout un tas de périphériques intégrés (port série, bus i2c, SPI, convertisseurs ADC, timers, GPIO, etc.). Le diagramme ci-contre extrait de la documentation (*datasheet*) Atmel montre clairement qu'il ne s'agit pas d'un simple processeur, mais d'un

système complet. Le processeur AVR lui, n'est qu'un élément, présenté en haut du schéma sous le terme « AVR CPU ».

Pour descendre encore plus profond dans le terrier du lapin blanc, il faudrait faire quelque chose qui n'est pas physiquement possible : casser le microcontrôleur en plusieurs morceaux (avec un marteau c'est faisable, mais ça marchera forcément moins bien ensuite). Une autre approche, plus réaliste, consiste à se tourner vers quelque chose qui utilise toutes ces briques séparément et qui à une certaine époque était la norme : un ordinateur familial.

Lorsqu'on regarde côte à côte le diagramme du microcontrôleur ATmega328P et celui d'un vénérable Commodore 64, ou de toutes autres machines de la même époque, on remarque immédiatement les similarités. Mais une différence ne transparaît pas sur les schémas : dans le cas de l'ordinateur familial des années 80, chaque élément prend la forme d'une puce. Ce que vous avez sous les yeux n'est pas l'intérieur d'un composant unique, mais tout le circuit imprimé de la machine.

Pour toucher du doigt et comprendre les relations entre ces briques de base, il nous suffit donc d'expérimenter exactement sur la même base, dans l'ordre d'importance des éléments : le processeur, la mémoire et les périphériques.

Ceci peut vous sembler être un défi insurmontable. Après tout, c'était des entreprises et des ingénieurs de talent qui avaient conçu ces machines, cela doit donc être inaccessible au commun des mortels, non ? En réalité, pas du tout. Bien avant l'arrivée de ces machines, des groupes d'enthousiastes existaient, constitués de personnes fabriquant leurs ordinateurs de cette manière. Certains de ces membres ont même fondé des sociétés qui existent toujours aujourd'hui, c'est le cas de Steve Wozniak, membre du *Homebrew Computer Club*, père de l'Apple I et co-fondateur d'Apple.

Aujourd'hui, une telle chose est à votre portée, une excellente source de connaissances et un moyen fantastique de renouer avec l'histoire de votre loisir, qu'il s'agisse d'électronique, de programmation ou de bidouille. Mieux encore, vous pouvez le faire à moindre coût, à votre rythme et avec relativement peu de matériel. Pour vos premiers pas, une platine à essais, un processeur, quelques résistances et leds, et quelque chose pour piloter l'expérimentation (une carte Arduino) seront suffisants...

Ce dernier point peut paraître surprenant, mais c'est l'une des raisons qui rendent cet exercice très facile aujourd'hui, en évitant d'avoir recours à un équipement très coûteux et difficile à utiliser. Une carte Arduino est un excellent moyen de créer un environnement pour expérimenter puisqu'elle permet de simuler des briques encore absentes au début du projet, comme une horloge ou la mémoire, tout en fournissant un moyen simple de « voir ce qui se passe ». Il est également possible, plus tard, qu'une carte Raspberry Pi soit également de la partie dans ce projet... Mais nous verrons cela en temps et en heure.

2. CHOISIR SON PROCESSEUR

Les ordinateurs familiaux des années 80 utilisaient différents processeurs en fonction de la marque et tantôt du modèle. Deux processeurs cependant se partageaient la vedette, les **Zilog Z80** (TRS-80, Sinclair ZX80, Amstrad CPC, Cambridge Z88, MSX, Philips VG5000, etc.) et les **MOS 6502/6510** (Commodore 64, Apple II, ACORN, Oric, BBC Micro, Nintendo NES, etc.). Il existe bien entendu d'autres modèles, comme le Motorola 6809, successeur du 6800 et source d'inspiration du 6502, qui équipait, par exemple le Thomson MO5 (pour les nostalgiques du « plan informatique pour tous »). Notez au passage que le Commodore 128 contenait non seulement un MOS 8502, une version compatible du 6502 capable de fonctionner jusqu'à 2 Mhz, mais également un Zilog Z80A, un Z80 capable de monter à 4 Mhz (plus un 6502 pilotant le lecteur de disquette pour le Commodore 128D).

Lorsqu'on se lance dans ce type de projet qui, je le rappelle, à terme, consiste tout de même à construire un ordinateur complet, le maître mot du choix du processeur est la quantité de documentation (vieille comme récente) disponible sur le sujet et la disponibilité des composants. Les 6502 (ou leurs dérivés les 6510) et les Z80 sont, dans ce domaine, les seules options intéressantes. Le Motorola 68000 (MC680000P12F plus précisément) est également un bon choix, mais uniquement pour un second projet, car ce processeur 16 bits est bien plus complexe à utiliser.

Au risque d'énerver quelques lecteurs puristes passionnés au passage, je dirais que Z80 et 6502 se valent du point de vue pédagogique. En termes de disponibilité cependant, le Z80 est bien plus facile à se procurer puisqu'on en trouve pour quelques euros sur eBay par exemple, y compris de la part de vendeurs français. Assurez-vous cependant de bien opter pour un modèle au format DIP-40 pouvant être placé sur une platine à essais et non une version plus moderne dans un format peu utilisable pour un hobbyiste (PLCC ou PQFP).

Nous nous baserons donc ici sur le Z80, mais tout ceci devrait être transposable sur un MOS 6502/6510/8502 ou n'importe quel autre processeur 8 bits avec quelques efforts et recherches.

Il existe plusieurs modèles de Z80, l'original à 2,5 Mhz, le Z80A à 4 Mhz, le Z80B à 6 Mhz et le Z80H à 8 Mhz. Bien entendu, il faut comprendre ici « jusqu'à » puisqu'il ne s'agit pas d'une fréquence imposée comme pour d'autres processeurs plus modernes (comme le Motorola 68000 par exemple). L'objectif est principalement ici de « voir » le processeur fonctionner et donc utiliser une cadence d'horloge humainement raisonnable, et contrôlable (par l'Arduino).

Vous trouverez très certainement plus facilement des Z80 et Z80A que d'autres modèles. Il est également possible que vous tombiez sur des processeurs Z80 ne provenant pas de chez Zilog, le constructeur original. Il s'agit simplement de clones fabriqués sous licence qui devraient fonctionner exactement de la même façon que l'original. Ce à quoi il faut faire attention, en revanche, est la présence d'annonces mentionnant « Z80 », mais ne concernant pas un processeur, mais un périphérique compatible :

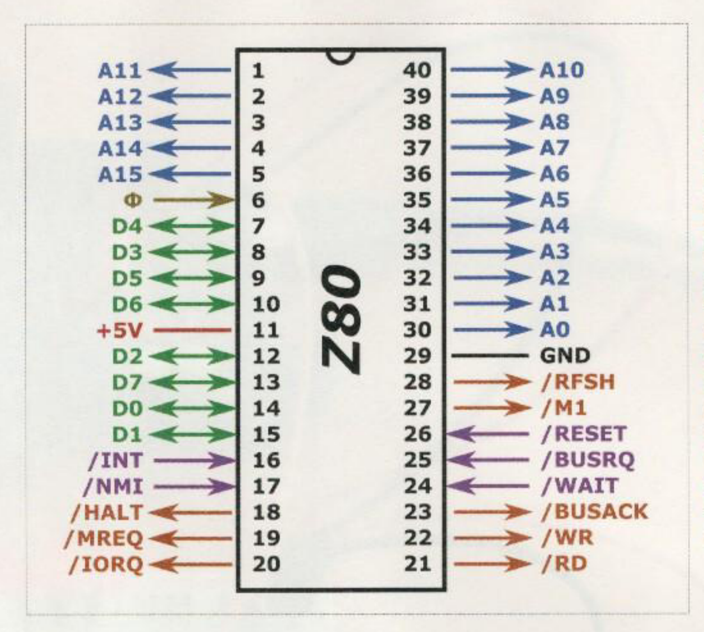
- Le CTC (*Counter/Timer Channel*) Z8430 qui fournit un *timer* programmable comme ceux intégrés dans le microcontrôleur de votre Arduino et qui gèrent **delay()** ou la PWM.
- Le PIO (*Parallel Input-Output*) Z8420, fournissant des ports d'entrée/sortie (GPIO), là encore, comme sur votre Arduino.
- Le SIO (*Serial Input Output*) Z8440 qui permet d'avoir un port série, comme le moniteur de... vous avez compris.
- Le DMA (*Direct Memory Access*) Z8410, un contrôleur permettant de provoquer des transferts directs entre deux périphériques.



Zilog n'a pas été le seul fabricant de processeurs Z80. Ici, un composant de chez STMicroelectronics sous la désignation Z84C00 qui s'avère être un Z80 amélioré compatible avec l'original, mais intégrant, en plus, les instructions compatibles avec Intel 8080A. Le brochage est parfaitement identique et il pourra être utilisé ici exactement comme un Z80 de chez Zilog.

- Le DART (*Dual Asynchronous Receiver Transmitter*) Z8470, un double port série.

Ces composants ne sont **PAS** des processeurs, qui eux sont sérigraphiés « Z8400 » ou « Z8400A » pour les modèles de chez Zilog, mais des périphériques qui peuvent s'y connecter et qui sont spécialement conçus pour le Z80. Lisez l'annonce, regardez la photo et cherchez le nom du composant s'il est spécifié dans l'annonce, il serait dommage de vous retrouver à mettre la charrue avant les bœufs...



Voici le brochage d'un Zilog Z80. Contrairement à ce que vous pourriez croire, toutes ces broches ne sont pas des entrées/sorties comme sur un ATmega328P, le Z80 n'en dispose pas. Ce sont des bus et des signaux, le Z80 est un processeur et non un microcontrôleur.

3. LE Z80 SUR PLATINE : LE B.A.BA

Et voilà ! Tout fébrile, vous avez décidé de vous lancer dans l'aventure, avez commandé votre ou vos processeurs (2 c'est mieux, on ne sait jamais) et ils viennent d'arriver par la poste. Il est temps de faire un premier essai pour vérifier que tout cela fonctionne et, par la même occasion, apprendre énormément de choses dont vous ne deviniez pas l'existence jusqu'alors...

Un processeur est juste une unité centrale de traitement ou **CPU** en anglais pour *Central Processing Unit*. Son travail est d'exécuter des **instructions** qui prennent la forme de valeurs numériques. L'ensemble de ces instructions compose un programme en **langage machine**. Une instruction est composée d'un **opcode** suivi ou non de valeurs qui forment des arguments. Exemple :

- l'instruction `0x00`, n'est composée que de l'opcode `NOP` (*No OPeration*), « ne rien faire », ayant pour valeur 0x00 ;
- l'instruction `0xc3 0x06 0x00`, se compose de l'opcode `JP` (comme *jump*) de valeur `0xc3` suivi de deux octets formant l'adresse à laquelle sauter (ici `0x0006`) ;
- l'instruction `0x3e 0x42` se compose de l'opcode `LD A` de valeur 0x3e et de l'argument `0x42`, et a pour effet de charger cette valeur dans le registre A (un registre est une minuscule zone mémoire interne au processeur, il en existe généralement plusieurs tantôt réservés à différents usages).

Comme il est relativement difficile de travailler directement en langage machine, on utilise une représentation plus intelligible appelée **assembleur**, composée d'instructions désignées de façon plus logique et lisible. Les trois instructions précédentes, en assembleur, s'écriraient respectivement :

- `NOP` ;
- `JP 0x0006` ;
- `LD A,0x42`.

Le fait de traduire l'assembleur en langage machine est la phase d'assemblage. Avec un système moderne, ceci est une étape de la compilation d'un programme, effectuée par l'**assembleur**. Vous ne le voyez pas en « vérifiant » un croquis Arduino, mais celui-ci est compilé pour devenir de l'assembleur puis assemblé pour devenir du langage machine avant d'être chargé dans la mémoire du microcontrôleur de votre carte (dans les grandes lignes).

Un processeur Z80 sur platine à essais cadencée et contrôlée par une carte Arduino UNO, la première étape d'un voyage d'exploration pédagogique avec plein de leds qui clignotent... tout le monde aime les leds qui clignotent !

Le Z80 utilise 252 instructions différentes de base permettant toutes sortes d'opérations, des sauts, des conditions, des tests, des additions, des soustractions, des lectures en mémoire, etc.

Le processeur, en brave exécutant ne fait que ce qu'on lui dit de faire sous forme d'instructions. Mais d'où viennent-elles puisque le processeur n'a aucune mémoire en lui-même ? Il les lit tout simplement sous la forme d'états qui lui sont présentés sur une série de broches : c'est le **bus de données** (broches 7, 8, 9, 10, 12, 13, 14 et 15 sur le Z80). À chaque cycle d'horloge, le processeur va lire l'état de ces 8 broches, en déduire un octet et l'utiliser comme une partie d'une instruction.

Si le processeur, par exemple lit `0xc3`, `0x06` puis `0x00`, il sait que la prochaine instruction devra être lue à l'adresse `0x0006`, soit au 7ème octet de la mémoire. Pour savoir où il se trouve à n'importe quel moment de l'exécution, il embarque un **compteur ordinal** également appelé **pointeur d'instruction** (le terme pointeur vous rappelle quelque chose ?), en anglais un *Program Counter* abrégé PC.

L'instruction en question ne fait rien d'autre que de dire au processeur que le compteur ordinal vaut

maintenant 0x0006. Reste ensuite à trouver un moyen pour faire en sorte que la mémoire sache quel octet présenter sur le bus de données. Pour cela, le processeur dispose de broches spécifiques formant également un bus : le **bus d'adresse** (broches 1 à 5 et 30 à 40). Les différents états des broches ou **lignes** d'adresse forment une valeur d'adresse et celle-ci est directement intelligible par le composant qui gère la mémoire.

Ainsi comme nous avons changé le **compteur ordinal** pour sauter à l'adresse 0x0006, le processeur va changer l'état des broches du bus d'adresse en conséquence puis, à nouveau lire le bus de données. Il récupère ainsi le début de la prochaine instruction, et le programme poursuit son exécution. Bien entendu, le **compteur ordinal** change également lui-même, et s'incrémente automatiquement en fonction de la lecture des instructions. En lisant 0xc3, l'opcode JP, le processeur sait qu'il doit encore lire deux fois le bus de donnée pour avoir l'instruction complète. La valeur du compteur ordinal est alors augmentée de 1, l'adresse présentée sur le bus d'adresse, et le bus de donnée à nouveau lu, et ainsi de suite.

Dans le cas du Z80, après initialisation, le compteur ordinal est à 0x0000, c'est là que le processeur s'attendra à avoir le premier octet d'une instruction à exécuter. D'autres processeurs fonctionnent de manière différente. Le MOS 6502, par exemple, présente sur le bus d'adresse 0xFFFC puis 0xFFFD et utilise ensuite les valeurs lues pour le compteur ordinal. Si les valeurs 0x10 et 0x00 sont successivement lues de cette manière, le compteur ordinal contiendra 0x0010 et le processeur lira le début de la prochaine instruction à cette adresse. On parle alors de vecteur de reset à l'adresse 0xFFFC/0xFFFD ou $FFFC/$FFFD (les valeurs hexadécimales sont souvent spécifiées avec $ et non 0x dans le domaine).

Notez au passage que le premier octet lu est celui le plus à gauche et le second le plus à droite de la valeur finale sur 16 bits. Le Z80 fonctionne de la même manière. On parle d'architecture *little endian* pour décrire cet ordre de lecture. Remarquez également que le bus de données utilise 8 broches et donc 8 bits. De plus, dans son fonctionnement interne, le processeur utilise également uniquement des valeurs sur 8 bits, les adresses sont coupées en deux (d'où le *little endian*). C'est cet état de fait qui explique le terme « processeur 8 bits », par opposition à, par exemple, un Motorola 68000, avec 16 bits de données et un fonctionnement interne utilisant des valeurs sur 16 bits (et des registres de 16 bits).

En résumé donc, on pourrait se dire que deux choses nous sont indispensables, un signal d'horloge pour cadencer tout cela et faire en sorte que le processeur exécute ses cycles, et de la mémoire pour y placer les instructions.

Le premier point est juste, mais pas le second. Une carte Arduino peut nous permettre de fournir un signal d'horloge à la vitesse de notre choix, il suffit de changer l'état d'une broche, mais également de simuler la mémoire. En effet, ceci se résume dans les grandes lignes à lire l'état de 16 broches du bus d'adresse et changer l'état des huit broches du bus de données. Quand le processeur demandera une donnée à une adresse spécifique, la carte Arduino pourra lui présenter sur le bus de données comme s'il s'agissait du contenu d'une mémoire.

Mais ceci est pour plus tard. Avant cela, nous pouvons faire un premier test avec simplement le signal d'horloge, quelques résistances et des leds.

4. LE Z80 SUR PLATINE : VERS LE PREMIER ESSAI

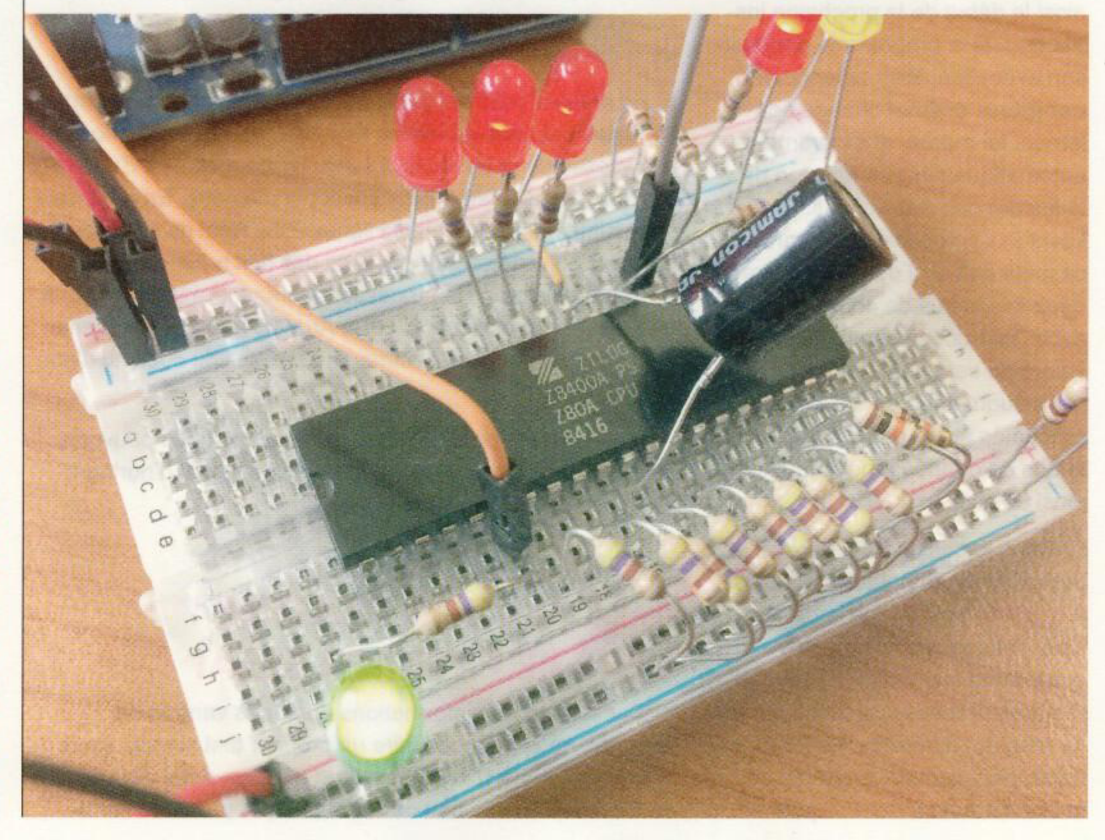
Sur la base des informations que nous venons de résumer, nous pouvons faire un premier test simple, sans mémoire. En effet, le Z80 s'attend à lire un octet sur son bus de données en fonction de l'adresse qu'il présente sur le

bus d'adresse. Si le bus de données est toujours à `0x00`, le processeur pensera lire l'instruction `NOP` à n'importe quelle adresse demandée. Pour cela, rien de plus simple, il nous suffit de mettre les 8 broches du bus à la masse via des résistances de rappel de 1 Kohm (ou 470 ohms, ça marche aussi).

Le bus d'adresse lui est unidirectionnel et n'a pas besoin d'être connecté pour l'instant, si ce n'est à trois leds via des résistances de 470 ohms et reliées à la masse. Nous n'avons pas besoin des 16 broches du bus d'adresse pour constater une activité, les trois premiers bits de l'adresse seront suffisants : A0 (30), A1 (31) et A2 (32). Notez au passage que ces sorties ne sont pas faites pour contrôler directement des leds, car le processeur n'est pas censé fournir autant de courant (ce ne sont pas des GPIO). C'est également pour cette raison que nous nous limitons à trois sorties/leds et non 8 ou 16. Nous verrons ultérieurement qu'il existe d'autres solutions pour visualiser l'activité du bus d'adresse.

Le processeur Z80 dispose également d'un certain nombre de broches en entrée que nous devons contrôler pour qu'il s'anime :

- /INT (16) : Le signal *INTerrupt Request* permet à un périphérique de demander une interruption du fonctionnement du processeur. Celui-ci va alors terminer l'exécution de l'instruction en cours et honorer, ou non, la demande selon la configuration en place. Le signal, comme tous ceux qui suivent, est actif à l'état bas (interruption demandée si la broche est mise à la masse). Nous n'en avons pas l'usage et mettons donc cette broche, comme

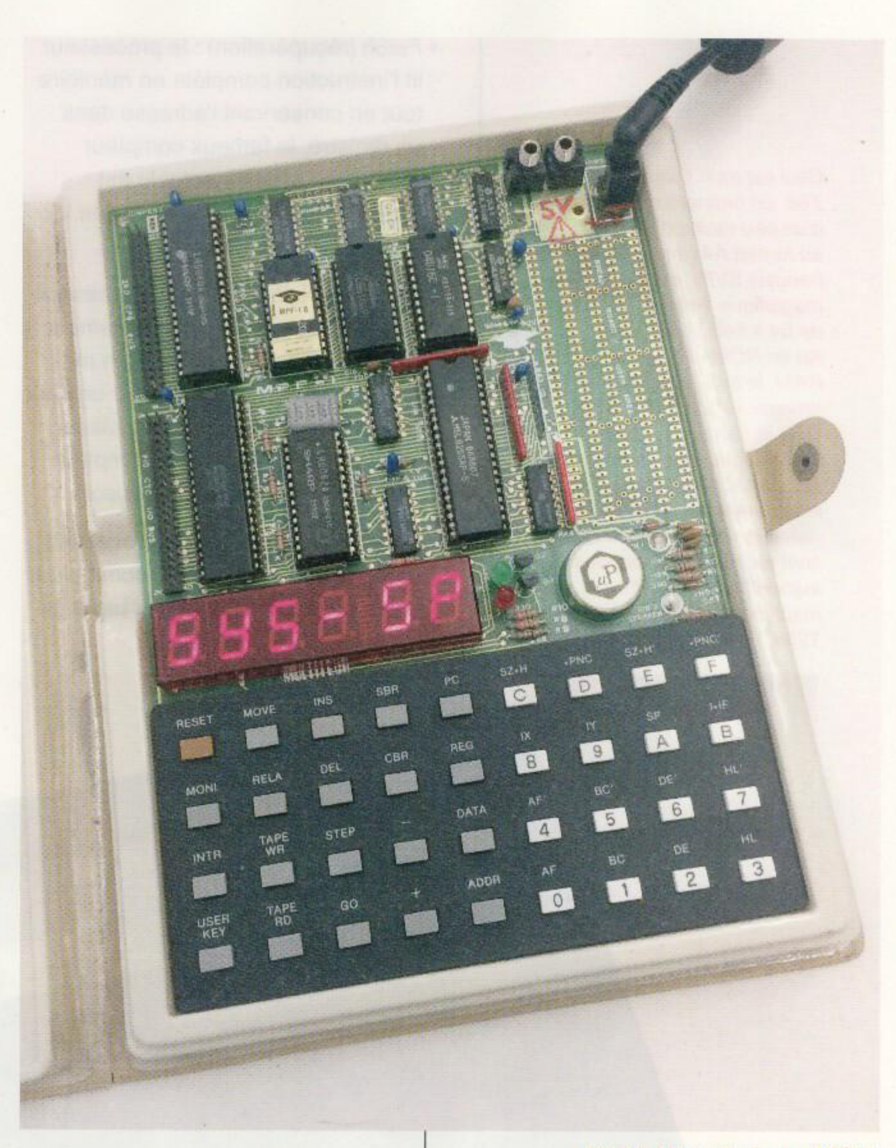


Ce processeur Zilog Z80A (marqué Z8400A) a été fabriqué la 16ème semaine de l'année 1984 (marquage « 8416 »), soit il y a plus de 33 ans. Non seulement il fonctionne encore, mais en plus il est toujours capable de nous apprendre énormément de choses et nous donner énormément de plaisir.

toutes les suivantes à la tension d'alimentation via une résistance de 1 Kohm.

- /NMI (17) : *NonMaskable Interrupt* est une version plus « agressive » du signal précédent. Là encore c'est une demande d'interruption, mais le processeur doit l'honorer après avoir fini l'exécution de l'instruction courante, sans possibilité de l'ignorer.
- /BUSRQ (25) : *BUS Request* est une demande de priorité encore plus élevée que le signal NMI. Si cette broche est passée à l'état bas le processeur va se déconnecter du bus d'adresse et des différents autres signaux/broches qu'il contrôle (/MREQ, /IORQ, /RD et /WR) après la fin du cycle courant. Ce signal permet donc de dire au processeur de « tout lâcher », car quelque chose d'autre veut accéder aux bus.
- /WAIT (24) : Demande au processeur d'attendre, car les informations sur le bus de données ou d'adresse ne sont pas encore prêtes.

Nous aurons donc besoin d'utiliser 4 résistances de rappel à la tension d'alimentation pour que le processeur considère ces signaux comme non actifs. Le Z80 dispose également de broches utilisées en sortie, mais toutes ne nous intéressent pas (pour l'instant). L'un des deux signaux qui vont nous permettre de voir que le processeur fonctionne, en plus des leds sur le bus d'adresse, est /M1 (27).



Voici le Micro-Professor MPF-I, un « ordinateur » pédagogique 8 bits historiquement fabriqué et commercialisé en 1981 par Multitech (maintenant Acer), construit autour d'un Zilog Z80 à 1.79 Mhz. Construire ce type de machine aujourd'hui prend du temps, mais est quelque chose de totalement accessible pour un amateur passionné. Le MPF-1B est encore fabriqué de nos jours par Flite Electronics qui en a racheté les droits à Acer en 1993, et vendu pour quelques £225 sur le Web.

Actif au niveau bas (d'où le « / » devant le nom), ce signal indique que le Z80 est dans la phase de lecture de l'opcode d'une instruction. « M1 » signifie *Machine Cycle One* (premier cycle machine) et pour interpréter le sens de ce signal, il faut comprendre le cycle de fonctionnement d'un processeur. Celui-ci est constitué de trois étapes :



Ceci est mon Cambridge Z88, un ordinateur portable d'un peu moins d'un kilo, au format A4 (avec clavier français SVP), avec un magnifique écran STN de 64 x 640 pixels, 128 Ko de ROM, 32 Ko de RAM, le tout animé par un processeur Z80A à 3.2768 MHz et fonctionnant avec un système d'exploitation appelé OZ. Ceci est littéralement une pièce de collection et des applications sont toujours développées aujourd'hui pour cette machine commercialisée en 1988.

- *Fetch* (récupération) : le processeur lit l'instruction complète en mémoire tout en conservant l'adresse dans un registre, le fameux compteur ordinal. Le premier octet lu est l'opcode, c'est à ce moment que la broche /M1 est active.
- *Decode* (décodage) : Le processeur analyse l'opcode et éventuellement récupère d'autres données en mémoire, comme l'adresse pour un saut (**JP**) ou encore une valeur à placer dans un registre (**LD**). Le compteur ordinal est incrémenté au besoin.
- *Execute* (exécution) : Le processeur fait ce qu'on lui demande comme par exemple changer l'état des broches sur le bus d'adresse pour récupérer le prochain opcode pour un saut, ou encore changer la valeur dans un registre.

Une fois tout cela fait, le cycle recommence. On parle de cycle d'instruction ou de cycle machine. Notez qu'un cycle ne correspond pas à un cycle d'horloge. En fonction des instructions exécutées, le nombre de cycles d'horloge (le nombre d'impulsions envoyées sur la broche dédiée) peut être plus ou moins important. Sur le Z80, un **NOP** par exemple, utilise 4 cycles d'horloge et un **JP** 10 cycles d'horloge, mais le traitement complet de ces instructions



correspond toujours à un cycle machine complet. En connectant /M1 à une led via une résistance et à la tension d'alimentation nous pourrons donc voir quand le processeur est à l'étape *fetch* et la led clignotera au rythme du cycle machine.

/RD (21) est un autre signal que nous pouvons observer de la même manière via une led connectée de la même façon. Ce signal actif à l'état bas est contrôlé par le Z80 pour signifier qu'il souhaite accéder à la mémoire. En pratique cette broche est reliée à la broche /OE (*Output Enable*) de la mémoire, lui demandant ainsi de prendre en compte l'adresse sur le bus et de présenter les données sur le bus de données. Ici, avec une led, ceci nous permettra de voir lorsque le processeur tentera de lire la mémoire.

Il ne nous reste que deux broches à contrôler pour finir le circuit. La broche 6 est le signal d'horloge qui n'est pas nommé dans la documentation, mais simplement désigné par la lettre phi de l'alphabet grec (en France un parti politique a choisi cette lettre comme logo, ce qui est assez drôle puisqu'ici c'est un signal littéralement fait pour faire marcher le processeur au pas et le soumettre). Ce signal d'horloge est une succession d'états hauts et bas permettant au Z80 d'être cadencé, c'est le cycle d'horloge. En fonction du modèle de Z80, il est possible de monter la fréquence de ce signal jusqu'à 8 Mhz, mais ici nous le contrôlerons bien en deçà avec une carte Arduino UNO (40 ms haut, 40 ms bas, soit environ 12 Hz). Notez que c'est la transition de l'état bas à l'état haut qui est significative, on parle de déclenchement sur le front montant.

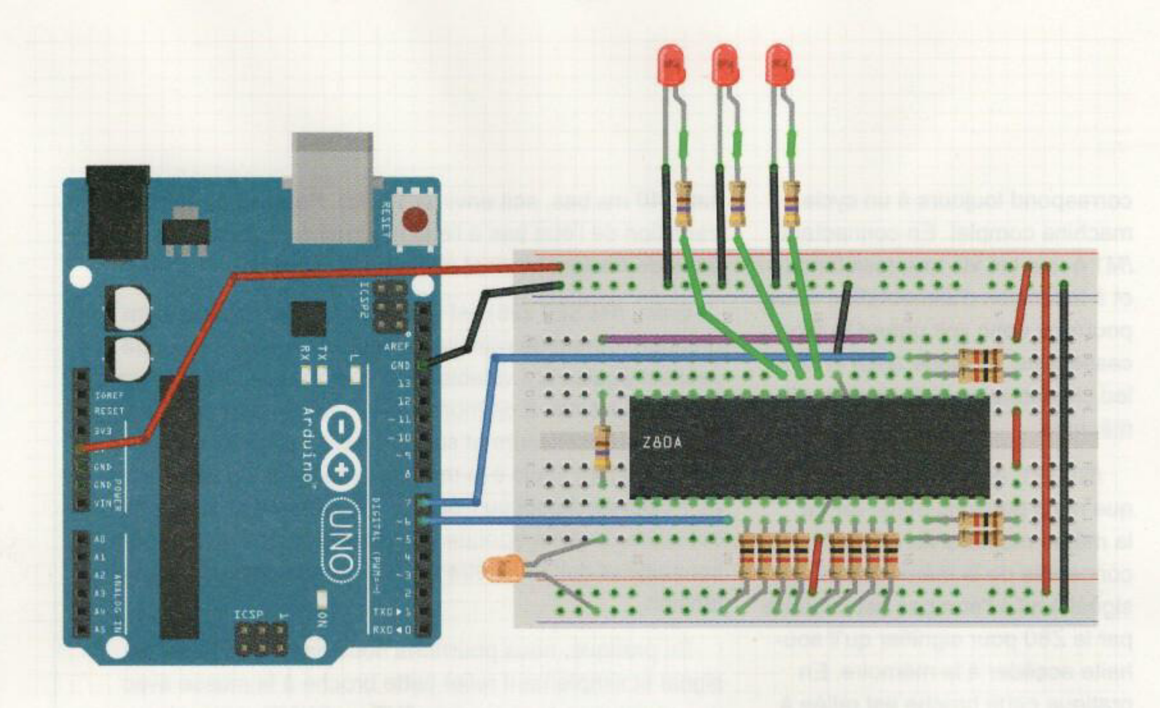
Enfin, /RESET (26) est le signal qui, comme son nom l'indique permet de réinitialiser le processeur. Notez que ceci est également valable pour la mise sous tension, car le Z80 se trouve, à ce moment-là, dans un état inconnu. Certains processeurs et surtout microcontrôleurs se réinitialisent tout seuls à la mise sous tension, on parle alors de *Power-on Reset* ou PoR. Le Z80 comme tous les processeurs de cette époque ne dispose pas de cette fonctionnalité et doit donc être réinitialisé à chaque mise sous tension.

En pratique, nous pourrions nous passer de gérer ce signal et simplement relier cette broche à la masse avec une résistance comme pour /INT ou /WAIT, mais mieux vaut faire les choses proprement. Ce signal sera donc piloté par la carte Arduino en accord avec la documentation du Z80 qui précise que la réinitialisation doit être faite en plaçant cette broche à la masse pendant au moins trois cycles d'horloge.

Enfin, il conviendra d'alimenter le Z80 en +5V appliqués sur la broche 11, avec la masse connectée sur la broche 29. Cette alimentation peut être ici fournie directement par la carte Arduino, mais il est très important d'avoir une alimentation parfaitement stable. Il est relativement facile de s'en assurer en plaçant un condensateur (~200 nF) entre l'alimentation et la masse, **au plus proche du Z80**. Une alimentation instable ou pleine de « bruit » est généralement la source de tous les maux et c'est donc de ce côté que vous devrez regarder en premier lieu en cas de problème. Placer le condensateur à cheval par-dessus le processeur sur la platine à essais est une excellente idée.

5. LE MONTAGE ET LE RÉSULTAT

Le montage sur platine à essais se résume à la mise en œuvre des explications précédentes sans le moindre changement. L'alimentation +5V et la masse sont fournies directement par la carte Arduino UNO. Les lignes /RESET et phi (l'horloge) sont respectivement connectées aux broches 7 et 6. Bien entendu, vous pourrez choisir arbitrairement les sorties qui vous arrangent le plus ici, vous n'aurez qu'à modifier le code en conséquence.



Le croquis associé à ce projet est, pour l'instant, très simple puisqu'il ne fait que cadencer le fonctionnement du Z80 et le réinitialiser :

```
#define CLKDELAY 40

#define B_CLOCK 6
#define B_RESET 7

// donne une impulsion sur la broche phi
void doClock(unsigned int n) {
  for(int i=0; i<n; i++) {
    digitalWrite(B_CLOCK, HIGH);
    delay(CLKDELAY);
    digitalWrite(B_CLOCK, LOW);
    delay(CLKDELAY);
  }
}

// provoque un reset en accord avec la doc
void doReset() {
  digitalWrite(B_RESET, LOW);
  doClock(5);
  digitalWrite(B_RESET, HIGH);
}

// configuration
void setup() {
  pinMode(B_CLOCK, OUTPUT);
  pinMode(B_RESET, OUTPUT);
  doReset();
}

// boucle principale
void loop() {
  doClock(1);
}
```

Pour faciliter les choses, nous commençons par déclarer deux fonctions. La première, `doClock()`, a pour tâche d'envoyer une impulsion sur la broche phi du Z80 avec des délais définis par la macro `CLKDELAY` en millisecondes. La valeur choisie ici est 40, ce qui laisse le temps de bien voir ce qui se passe tout en essayant de faire en sorte que ce ne soit pas trop ennuyant. Cette fonction prend en argument le nombre d'impulsions à envoyer.

La seconde fonction, `doReset()`, provoque une réinitialisation « propre » du Z80 en plaçant la ligne /RESET à la masse et en envoyant 5 signaux d'horloge (la documentation parle de 3 minimum), avant de faire revenir la broche à l'état haut.

On passe ensuite à notre fonction de configuration (`setup()`) configurant classiquement les sorties et provoquant la réinitialisation avec `doReset()`, puis à la boucle principale se contentant se fournir une impulsion d'horloge à l'infini.

Une fois le croquis chargé, si tout est correctement connecté (et l'alimentation stable), le Z80 devrait s'animer. À la vitesse utilisée ici, vous devrez être en mesure de voir /M1 s'activer juste un peu avant /RD de façon régulière (et plus lentement que le signal d'horloge) et surtout constater une incrémentation en binaire sur les trois leds du bus d'adresse : 000, 001, 010, 011, 100, 101, 110, 111, 000, 001...

Ceci montre clairement que le processeur fait son travail, tente de lire la mémoire à partir de l'adresse `0x0000`, récupère l'opcode `NOP`, exécute l'instruction, incrémente le compteur ordinal, puis passe à l'instruction suivante et ainsi de suite. Comme nous ne visualisons que les trois premiers bits de l'adresse, nous avons l'impression que le cycle se répète. C'est effectivement le cas, mais uniquement une fois que les 65536 adresses mémoire auront été parcourues. 16 bits = 2^16 combinaisons = 65536 adresses possibles ou, en d'autres termes, une mémoire adressable de 65536 octets ou 64 Ko.

6. CE QUE NOUS AVONS APPRIS ET LA SUITE

Le résultat obtenu peut paraître, pour un œil non averti, quelque chose de relativement peu d'intérêt. Après tout, il ne s'agit que de quelques leds qui clignotent et un tel effet peut être obtenu facilement avec une carte Arduino seule. Mais c'est le fait que vous connaissiez le sens de ces clignotements qui donne toute son l'importance au résultat : vous faites fonctionner un processeur dont la conception remonte à 1976 et qui est toujours utilisé aujourd'hui dans les calculatrices Texas Instruments par exemple (TI-83 Premium CE, TI-82 Advanced, etc.). Une véritable légende !

Mieux encore, vous avez sous les yeux quelque chose que vous ne pourrez jamais voir avec un microcontrôleur comme l'AVR d'un Arduino : son fonctionnement interne. Nous avons appris ici ce qu'est un bus d'adresse et de données, le principe basique de fonctionnement d'un processeur, les signaux qu'il doit prendre en compte, et les cycles qui font que ce fonctionnement est possible. Nous avons même, en quelque sorte, écrit notre premier programme que nous avons assemblé à la main. Certes il n'est constitué que d'un seul opcode, mais nous l'avons physiquement fourni au processeur avec nos petits doigts musclés...

Je n'aime pas faire des articles à suite étalés sur plusieurs numéros, mais ici nous n'avons malheureusement pas le choix. Après tout, nous venons de débuter un voyage qui comprend de nombreuses étapes et dont la destination finale se résume à la construction d'un ordinateur des années 80.

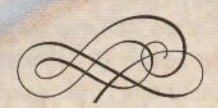
La suite logique dans ce genre d'expérimentations consiste normalement à construire un circuit d'horloge cadençant le processeur (avec un NE555 ou un oscillateur à quartz par exemple) et à associer une mémoire (généralement une UVPROM) au Z80. Ceci cependant nécessite d'autres composants et surtout du matériel, comme un programmeur d'EPROM et un effaceur UV.

Une autre approche est cependant possible et plus pédagogique : reposer sur la carte Arduino, du moins encore quelque temps, avant de passer à quelque chose « en dur ». Si la carte Arduino est en mesure de se comporter comme une mémoire et donc de lire une adresse sur tout ou partie du bus, alors elle sera également capable de présenter des données en conséquence. Nous pouvons donc nous servir du croquis non seulement comme source d'horloge et de réinitialisation, mais également comme mémoire et ainsi faire exécuter notre premier vrai programme au Z80. Voilà précisément quel sera l'objet du prochain article. **DB**

PRENEZ DE BONNES MESURES !

ET SI DÈS LA RENTRÉE ON SE METTAIT AU COURANT ?

Éric Le Bras



Le domaine du hack rime souvent avec la nécessité de répondre à un besoin nouveau par le biais d'un bricolage rapide. Cette démarche nous conduit spontanément à explorer de nouveaux territoires, à nous immerger dans le monde des techniques électroniques au-delà de la réflexion initiale. Au départ, le cadre d'analyse s'articulait essentiellement autour de la détection de la mise en fonctionnement d'une pompe à l'aide d'un montage Arduino. Cependant, pour appréhender pleinement ce sujet, un détour s'est vite imposé vers la route de la (re-)découverte de certaines lois visitées lors de notre passage au lycée. Un monde inconnu demeurait face à moi : celui de la détection et de la mesure du courant. Cet article vous propose un panorama des techniques et composants permettant de détecter et mesurer les courants et tensions électriques.

Détecter le courant circulant dans un circuit, c'est bien joli, mais de quoi parle-t-on précisément ? Lecteur de *Hackable*, vous possédez probablement déjà les concepts permettant de comprendre le phénomène électrique et les principales grandeurs s'y attachant : courant, tension, puissance, résistance. Si ce n'est pas le cas, que vous preniez le train en marche, ou que simplement vous souhaitiez rafraîchir ces notions, je ne peux que vous encourager chaudement à lire – ou relire – l'excellent article consacré à la Loi d'Ohm publié dans le numéro 1 de votre magazine. Dans la suite de cet article, nous appliquerons ces notions, ainsi que quelques autres permettant de comprendre les courants alternatifs.

Mais d'abord, qu'entend-on exactement par mesurer une tension ou un courant ? À quelle grandeur s'intéresser et pour quel besoin ?

Évidemment, le quoi dépendra fortement du pourquoi. S'il s'agit de vérifier le niveau d'une batterie, il suffira de mesurer la tension entre ses deux bornes. Mais si l'on veut savoir si un appareil est en fonctionnement, c'est à la mesure du courant que nous nous intéresserons. Et si nous souhaitons connaître la quantité d'énergie consommée – et accessoirement ce que cela va nous coûter –, nous souhaiterons alors mesurer la puissance.

1. MESURER LA TENSION

Commençons par le plus simple. L'Arduino est pourvu d'entrées analogiques (A0 à A5 sur un Uno) servant précisément à cela.

Pour mesurer une tension située dans la plage de l'entrée (0 à +5 volts), rien de plus simple : relions le dipôle (la batterie ou le composant électronique alimenté) aux bornes duquel nous souhaitons mesurer la tension, à l'entrée analogique, en respectant la polarité : le pôle + sur l'entrée analogique A0 à A5, le pôle - sur la masse GND.

Si la tension à mesurer est supérieure à 5 volts (tout en restant dans des limites ne vous mettant pas en danger, en gros jusqu'à 30 volts pour un courant continu), on peut utiliser un pont diviseur de tension constitué de deux résistances, ou bien un potentiomètre, pour ramener la tension dans la plage de 0 à 5 volts.

Le principe du pont diviseur est assez simple. Il s'appuie sur la loi d'additivité des tensions, qui précise que la tension

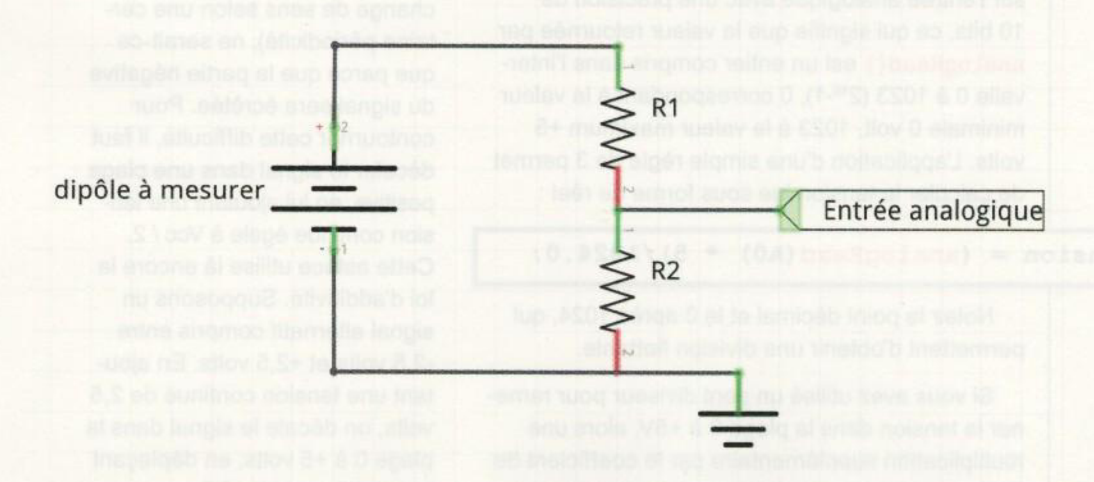


Fig.1 : Schéma d'un pont diviseur. Le dipôle peut être indifféremment un générateur (batterie, transformateur, etc.), ou bien un composant dans un circuit alimenté.

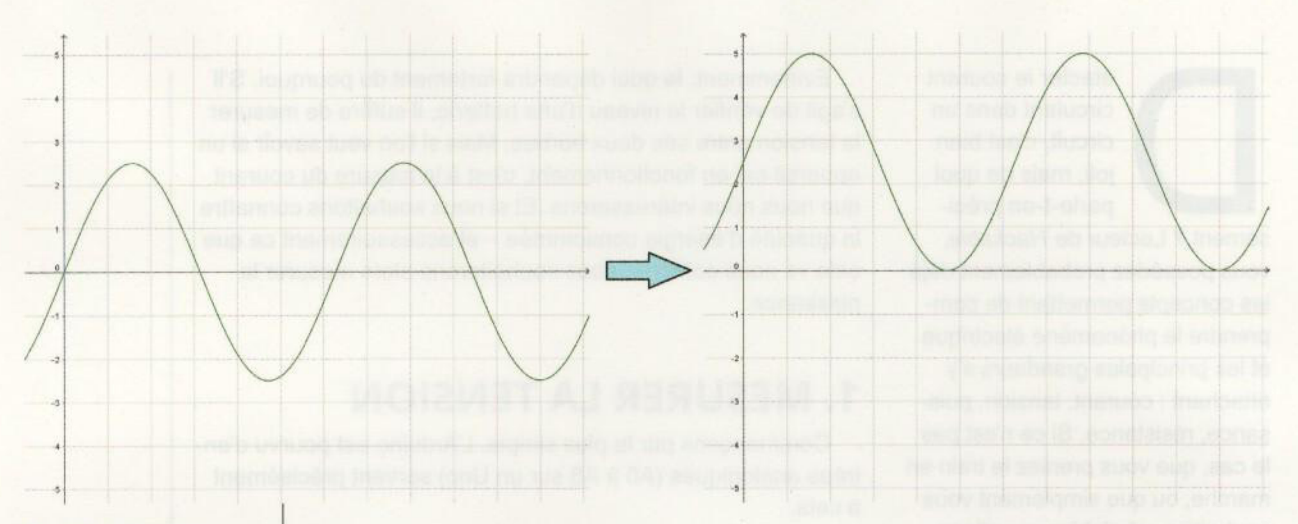


Fig. 2 : En ajoutant une tension de 2,5 volts, on décale tout le signal vers le positif. Ici, un signal alternatif de 2,5 volts (alternant entre -2,5 et +2,5 volts) devient un signal positif compris entre 0 et +5 volts.

aux bornes d'un ensemble de récepteurs branchés en série correspond à la somme des tensions de chacun d'entre eux. En particulier, si R1 = R2, alors la tension aux bornes de R1 est égale à la tension aux bornes de R2, et aussi égale à la moitié de la tension à mesurer. Précisons qu'un potentiomètre peut être vu comme un pont diviseur, dont le rapport entre les deux résistances pourrait varier de façon continue en fonction de la position du curseur.

La lecture de l'entrée analogique s'effectue au moyen de la fonction **analogRead()**, avec en paramètre l'identifiant de l'entrée analogique à lire. Le convertisseur analogique-numérique (ADC) effectue un échantillonnage de la tension présente sur l'entrée analogique avec une précision de 10 bits, ce qui signifie que la valeur retournée par **analogRead()** est un entier compris dans l'intervalle 0 à 1023 (2^{10}-1), 0 correspondant à la valeur minimale 0 volt, 1023 à la valeur maximum +5 volts. L'application d'une simple règle de 3 permet de calculer la tension lue sous forme de réel :

```
tension = (analogRead(A0) * 5)/1024.0;
```

Notez le point décimal et le 0 après 1024, qui permettent d'obtenir une division flottante.

Si vous avez utilisé un pont diviseur pour ramener la tension dans la plage 0 à +5V, alors une multiplication supplémentaire par le coefficient de division nous donnera la tension à mesurer.

En ce qui concerne l'entrée analogique de l'Arduino, comme nous l'avons vu sa plage d'utilisation est comprise entre 0 et +5 volts. Il n'est pas recommandé d'appliquer une tension négative (gare aux erreurs de polarité !) sans risque pour l'entrée analogique, voire pour le microcontrôleur. En pratique, des diodes de coupure protègent l'entrée, ce qui limite nettement le risque. Il n'est toutefois pas recommandé de lire une tension alternative directement (une tension alternative change de sens selon une certaine périodicité), ne serait-ce que parce que la partie négative du signal sera écrêtée. Pour contourner cette difficulté, il faut décaler le signal dans une plage positive, en lui ajoutant une tension continue égale à Vcc / 2. Cette astuce utilise là encore la loi d'additivité. Supposons un signal alternatif compris entre -2,5 volts et +2,5 volts. En ajoutant une tension continue de 2,5 volts, on décale le signal dans la plage 0 à +5 volts, en déplaçant le point 0 vers Vcc / 2.

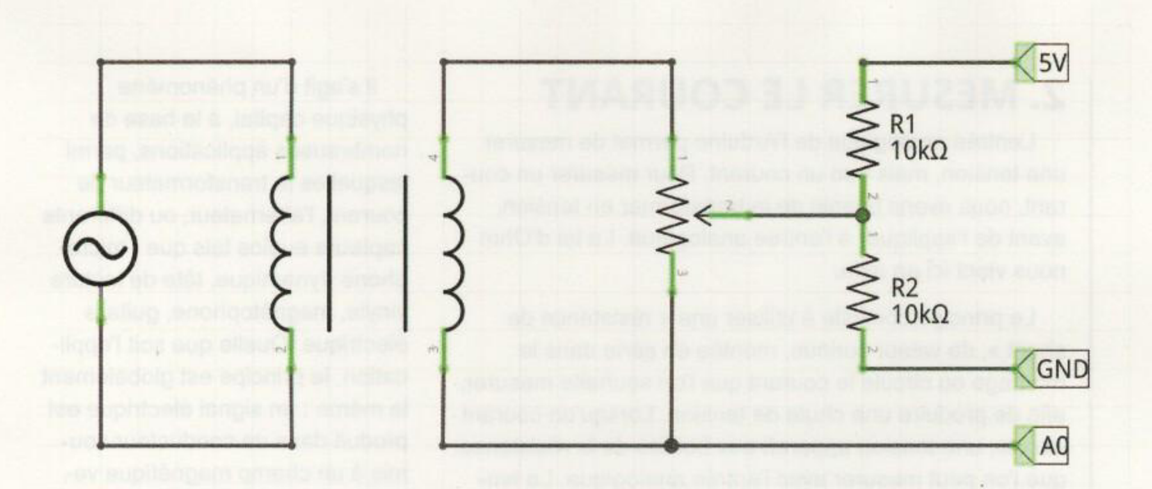


Fig. 3 : Mesure d'une tension alternative. Le transformateur est raccordé en parallèle avec le circuit électrique dont on souhaite mesurer la tension.

Cette fois-ci le pont diviseur utilisera deux résistances de valeur identique (10 KΩ ou plus) pour diviser par 2 la tension Vcc de l'Arduino (+5 V), et ainsi fournir la tension de 2,5 V à ajouter au signal d'entrée.

Si la tension alternative dépasse 12 volts, pour isoler le microcontrôleur on utilisera obligatoirement un transformateur pour ramener la tension dans cette plage, puis si nécessaire un pont diviseur ou un potentiomètre, avant de décaler le signal dans la plage 0 à +5 volts.

Selon le but recherché, il se peut que seule la valeur absolue de la tension alternative nous intéresse (voir plus loin dans cet article le chapitre consacré à la mise en pratique du capteur CT). Cela peut être le cas si l'on souhaite mesurer de façon simple la valeur d'une tension alternative sinusoïdale. La technique consiste à ne conserver que la partie positive du signal, en insérant une diode en série ou en parallèle dans le circuit de mesure. Ce composant électronique ne laissant passer le courant que dans un sens, le résultat obtenu est que la partie négative du signal alternatif est écrêtée.

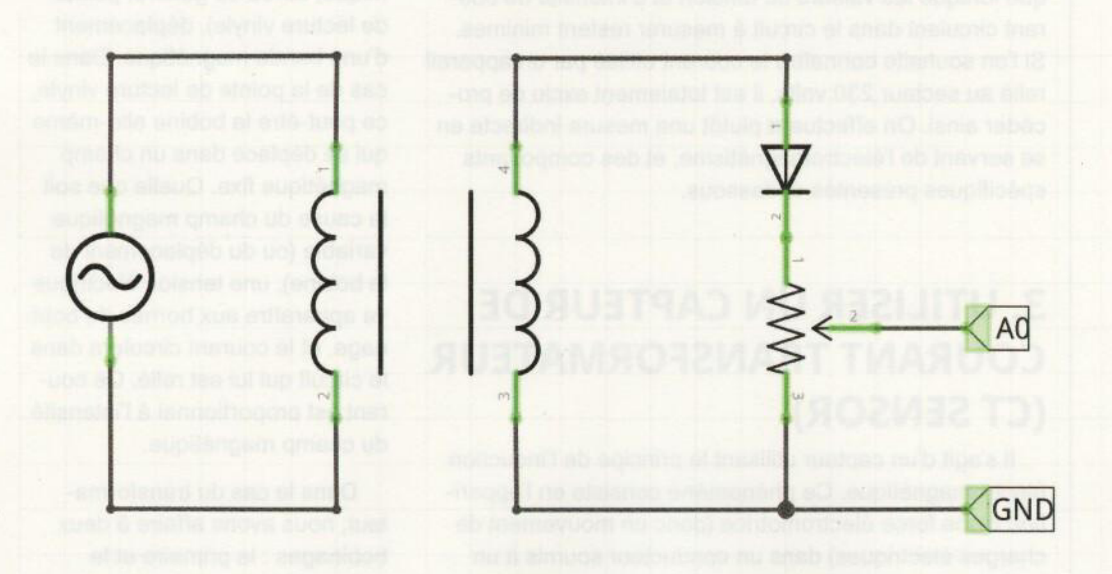


Fig. 4 : La diode écrête la partie négative du signal, en ne laissant passer le courant que dans un seul sens. Le triangle indique le sens du passage du courant.

2. MESURER LE COURANT

L'entrée analogique de l'Arduino permet de mesurer une tension, mais non un courant. Pour mesurer un courant, nous avons besoin de le transformer en tension, avant de l'appliquer à l'entrée analogique. La loi d'Ohm nous vient ici en aide.

Le principe consiste à utiliser une « résistance de shunt », de valeur connue, montée en série dans le montage où circule le courant que l'on souhaite mesurer, afin de produire une chute de tension. Lorsqu'un courant circule, une tension apparaît aux bornes de la résistance, que l'on peut mesurer avec l'entrée analogique. La tension et la résistance étant connues, la loi d'Ohm nous fourni la valeur du courant :

I = U / R

Cependant, la tension mesurée correspond à une chute de tension dans le circuit, ce qui aura une incidence sur son fonctionnement. Plus la résistance est élevée, plus la chute de tension est importante. A contrario, une résistance basse correspond donc à une chute de tension faible, donc une incidence moindre sur le circuit, mais une tension plus faible, donc une mesure moins précise. Dans certains cas, une amplification pourra même être nécessaire ! Bien sûr, la mesure d'un courant en insérant une résistance de shunt directement reliée à l'Arduino dans le circuit à mesurer ne pourra être mise en œuvre que lorsque les valeurs de tension et d'intensité du courant circulant dans le circuit à mesurer restent minimes. Si l'on souhaite connaître le courant utilisé par un appareil relié au secteur 230 volts, il est totalement exclu de procéder ainsi. On effectuera plutôt une mesure indirecte en se servant de l'électromagnétisme, et des composants spécifiques présentés ci-dessous.

3. UTILISER UN CAPTEUR DE COURANT TRANSFORMATEUR (CT SENSOR)

Il s'agit d'un capteur utilisant le principe de l'induction électromagnétique. Ce phénomène consiste en l'apparition d'une force électromotrice (donc un mouvement de charges électriques) dans un conducteur soumis à un champ magnétique variable.

Il s'agit d'un phénomène physique capital, à la base de nombreuses applications, parmi lesquelles le transformateur de courant, l'alternateur, ou différents capteurs audios tels que : microphone dynamique, tête de lecture vinyle, magnétophone, guitare électrique. Quelle que soit l'application, le principe est globalement la même : un signal électrique est produit dans un conducteur soumis à un champ magnétique variable, ou bien par le déplacement du conducteur lui-même dans un champ magnétique fixe. Concrètement, on utilise un bobinage de fil de cuivre enroulé autour d'un noyau (ferrite ou tôles feuilletées). L'ensemble est placé dans un champ magnétique variable. Selon l'application, le champ est dû à un phénomène physique variable : mise en rotation d'aimants (alternateur), circulation d'un courant alternatif (transformateur), mise en vibration d'une pièce métallique (diaphragme d'un micro dynamique, corde de guitare, pointe de lecture vinyle), déplacement d'une bande magnétique. Dans le cas de la pointe de lecture vinyle, ce peut-être la bobine elle-même qui se déplace dans un champ magnétique fixe. Quelle que soit la cause du champ magnétique variable (ou du déplacement de la bobine), une tension électrique va apparaître aux bornes du bobinage, et le courant circulera dans le circuit qui lui est relié. Ce courant est proportionnel à l'intensité du champ magnétique.

Dans le cas du transformateur, nous avons affaire à deux bobinages : le primaire et le secondaire. Les caractéristiques

du courant induit dépendent des grandeurs du courant appliqué au primaire, et du rapport entre le nombre de spires des deux bobinages. Pour un transformateur idéal (sans pertes), ces rapports s'établissent ainsi :

- **U1/U2 = N1/N2** (le rapport entre les tensions primaires et secondaires est égal au rapport entre les nombres de spires dans les bobinages primaires et secondaires).
- **I2/I1 = N1/N2** (le rapport entre les courants primaires et secondaires est l'inverse du rapport entre les nombres de spires dans les bobinages primaires et secondaires).

Il s'ensuit donc que l'intensité du courant traversant le circuit côté secondaire est inversement proportionnelle au rapport du nombre de spires entre les bobinages primaires et secondaires.

Notre capteur de courant fonctionne sur le même principe. Le champ magnétique variable est produit par le courant circulant dans un conducteur (le câble alimentant la lampe ou le PC). Le conducteur traverse un anneau de ferrite autour duquel est enroulé un bobinage unique (attention : le conducteur peut être indifféremment la phase ou le neutre, mais vous ne devez faire passer qu'un seul des deux dans le capteur). L'ensemble constitué du conducteur traversant le capteur peut être assimilé à un transformateur avec le conducteur jouant le rôle du primaire, et le bobinage du capteur celui du secondaire. On a donc :

- N1 = 1
- N2 = le nombre de spires du capteur

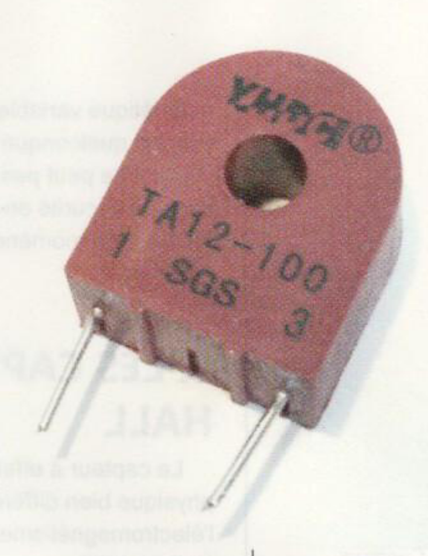
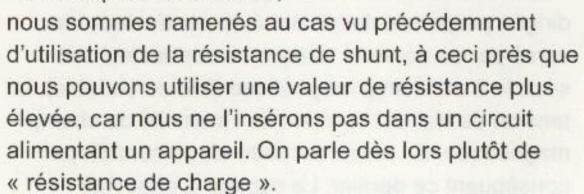


Fig. 5 : Un capteur de courant TA12-100. On le trouve souvent monté sur une breakout board, avec une simple résistance de charge, pour deux fois le prix du composant seul comme ici.

Le capteur TA12-100 que nous avons utilisé, avec 1000 spires, traversé par un conducteur, constitue un transformateur de rapport 1/1000. Ce capteur va donc produire un courant dont l'intensité sera égale à 1/1000 de l'intensité à mesurer circulant dans le conducteur. Comme notre Arduino ne mesure que des tensions, nous sommes ramenés au cas vu précédemment d'utilisation de la résistance de shunt, à ceci près que nous pouvons utiliser une valeur de résistance plus élevée, car nous ne l'insérons pas dans un circuit alimentant un appareil. On parle dès lors plutôt de « résistance de charge ».



La valeur de la résistance de charge sera choisie en fonction de l'intensité du courant à mesurer et du nombre de spires du capteur, afin d'obtenir une tension aux bornes de la résistance correspondant à la plage de mesure. Selon le but recherché, on « décalera » ensuite la tension alternative autour de Vcc / 2 (2,5 volts dans le cas de l'Arduino), ou bien on écrêtera la partie négative à l'aide d'une diode.

Le grand intérêt de ce capteur est son fonctionnement non invasif et totalement sécurisé, puisque qu'il permet l'isolation galvanique entre le circuit à mesurer et le capteur (il n'existe aucune liaison conductrice entre eux). Il est enfin très simple à mettre en œuvre.

Un défaut mineur réside toutefois dans le fait que le noyau se magnétisant avec l'usage, ceci dans le temps introduit un biais dans la mesure, biais qui ne pourra être compensé qu'en démagnétisant le noyau (on constate d'ailleurs ce même phénomène avec les micros).

Notons toutefois que ce capteur reposant sur le phénomène de l'induction électromagnétique, il ne permet de détecter que des courants alternatifs (souvenez-vous : le courant induit est dû à un champ

magnétique variable). Si l'on souhaite mesurer un courant quelconque (alternatif ou continu), et si ce courant ne peut pas être mesuré dans des conditions de sécurité en utilisant la résistance de shunt, un autre phénomène physique vient à notre aide : l'effet Hall.

4. LES CAPTEURS À EFFET HALL

Le capteur à effet Hall repose sur un phénomène physique bien différent, quoique toujours relié à l'électromagnétisme, découvert par Edwin Herbert Hall en 1879.

L'effet Hall consiste en l'apparition d'une tension dirigée perpendiculairement à un conducteur traversé par un courant, lorsque ce conducteur est soumis à un champ magnétique (fixe ou variable). La tension de Hall est fonction de l'intensité du champ magnétique. En mesurant celle-ci, on connaît par conséquent ce dernier. Le capteur à effet Hall mesure donc un flux magnétique, et non le courant induit par sa variation, comme dans le cas du transformateur. Autre différence : un capteur basé sur l'effet Hall nécessite un courant d'alimentation.

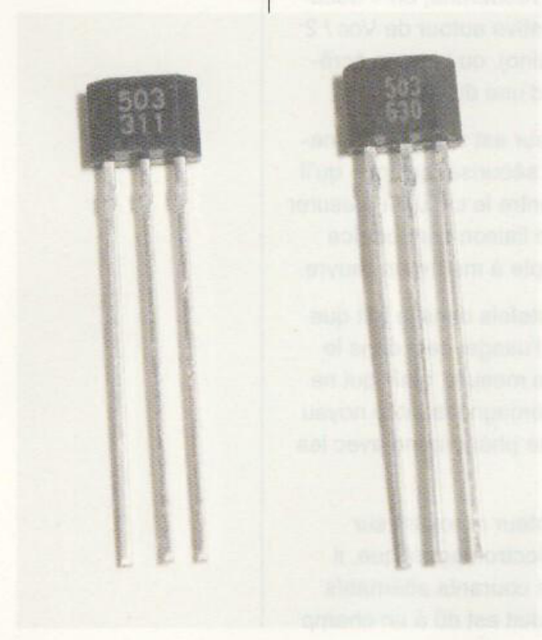


Fig. 6 : Capteurs à effet Hall. Il s'agit de modèles plutôt destinés à détecter le champ magnétique d'un aimant, mais ils peuvent également être mis en œuvre avec un tore de ferrite pour constituer un capteur très sûr, sans connexion électrique directe avec le courant à mesurer.

Les capteurs à effet Hall étant ainsi capables de détecter un flux magnétique quelconque en le transformant en tension, ils trouvent de très nombreuses applications, notamment celles basées sur la détection d'aimant. Ainsi, ils permettent de créer des détecteurs d'ouverture, de vitesse, de fonctionnement de moteur ou de niveau de liquide, pour ne citer que les plus usuels. Ces capteurs, n'étant soumis à aucun effort mécanique, sont d'une très grande fiabilité.

Par ailleurs, le champ magnétique pouvant être produit non plus par un aimant, mais par un courant électrique, dans le cas qui nous intéresse l'effet Hall permet également de concevoir des détecteurs de courant. Contrairement au phénomène d'induction vu précédemment, le phénomène se produit dans tous les cas de présence d'un champ magnétique, variable ou pas. Un capteur basé sur ce phénomène physique est donc – contrairement au capteur de type transformateur – apte à mesurer un courant, quel qu'il soit, étant donné que tout courant électrique, même continu, produit un champ magnétique autour du conducteur qu'il traverse.

Les capteurs à effet Hall se présentent généralement sous la forme d'un boîtier de quelques millimètres de côté. Pour détecter un courant, le capteur est parfois inséré dans un tore de ferrite.

Attention : Une sonde ampèremétrique fonctionnant sur ce principe peut être fabriquée à la maison. Si vous souhaitez aller plus loin sur le sujet, je vous recommande la lecture de cette excellente page de Yann Guidon : http://ygdes.com/sonde-courant/.

Les capteurs à effet Hall se déclinent dans de très nombreuses variantes, toutefois ceux permettant de mesurer un courant possèdent généralement trois connexions :



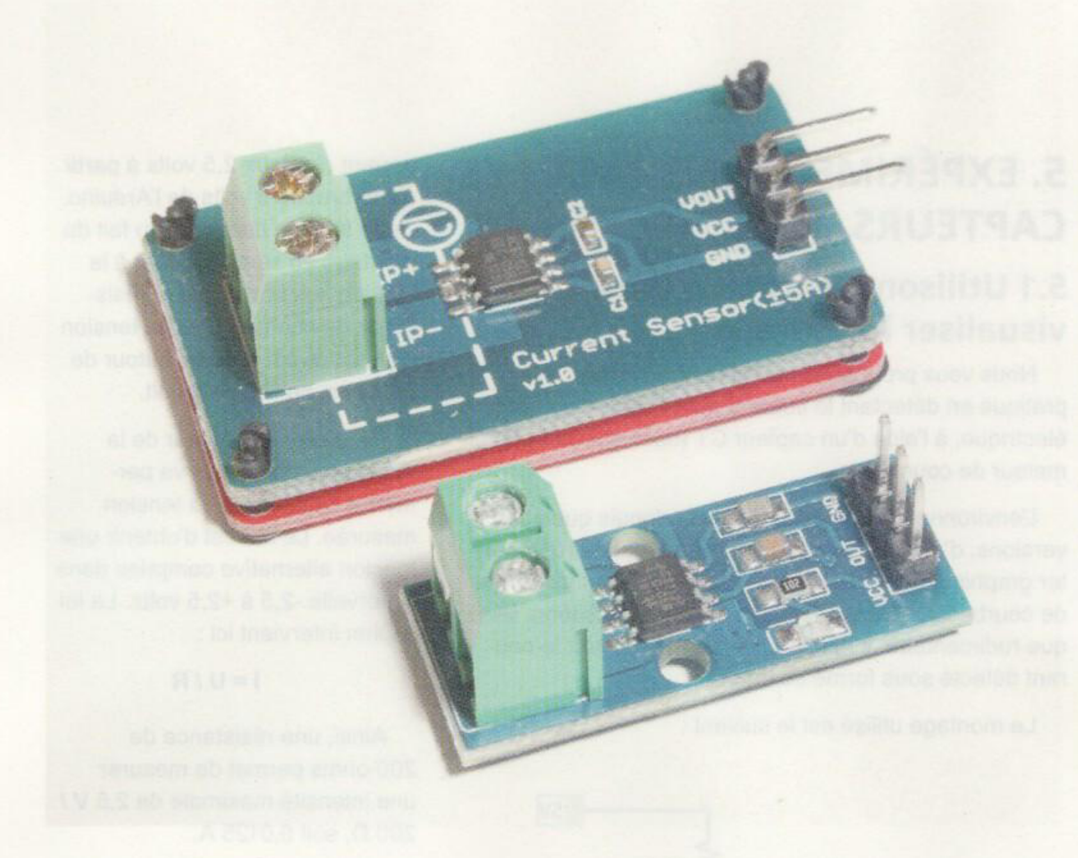


Fig. 7 : Ces deux modules sont à base d'ACS712. Le plus gros avec une plaque d'isolation (en haut) est celui qui a donné les meilleurs résultats lors de nos expérimentations (voir la mise en pratique plus loin dans cet article). On le trouve pour un peu moins de 5 euros chez Deal Extreme (www.dx.com).

- Vcc est reliée à une source de tension (broche +5 volts sur l'Arduino).
- GND est reliée à la masse.
- La sortie Vout varie linéairement avec le courant (alternatif ou continu) capté en entrée. Lorsque le capteur est au repos (pas de courant détecté), Vout présente une tension égale à Vcc/2. Lorsque le capteur détecte un courant, la différence mesurée sur cette broche en plus ou en moins par rapport à Vcc/2 est proportionnelle au courant détecté. Le coefficient à appliquer pour convertir la tension en courant correspond à la sensibilité en mV/A fournie par la *datasheet*.

Notons également l'existence de l'ACS712, un détecteur de courant sous forme de circuit intégré utilisant l'effet Hall. Ce capteur nécessite une connexion électrique avec le circuit à mesurer. On le trouve souvent monté sur *breakout board*, avec un bornier pour les connexions vers le circuit à mesurer. Ce capteur plutôt économique peut trouver des applications s'il est utilisé sur une installation fixe dans des conditions de sécurité satisfaisantes (du fait des connexions électriques nécessaires) : isolation, mais aussi respect du courant maximal spécifié par le composant.

Bien d'autres solutions existent, mais la place – et le temps ! – manqueraient pour les citer toutes. Mentionnons toutefois encore l'INA219 de Texas Instruments. Il s'agit d'un circuit intégré nettement plus complexe, intégrant de la logique et avec des paramètres programmables, qui a pour particularité de mesurer en même temps le courant et la tension. Il peut même multiplier l'un par l'autre pour fournir la puissance en direct ! Ce circuit est un vrai périphérique doté d'une interface I2C pour communiquer avec le microcontrôleur. Comme il ne peut mesurer que des tensions comprises entre -26 et +26 volts, son domaine d'application est destiné au contrôle de tension ou de consommation au sein d'appareils utilisant des tensions comprises dans cette gamme. On le trouvera donc dans les équipements informatiques ou télécoms, alimentations, chargeurs de batteries, fers à souder, etc.).

5. EXPÉRIMENTONS AVEC LES CAPTEURS

5.1 Utilisons le traceur pour visualiser le courant

Nous vous proposons maintenant de passer à la pratique en détectant le courant utilisé par un appareil électrique, à l'aide d'un capteur CT (basé sur un transformateur de courant).

L'environnement Arduino dispose, depuis quelques versions, d'un « traceur série » permettant de représenter graphiquement des valeurs numériques, sous forme de courbes. Ce traceur est très simple d'utilisation. Bien que rudimentaire, il nous suffira pour visualiser le courant détecté sous forme de courbe.

Le montage utilisé est le suivant :



Fig. 8 : Le signal en sortie du TA12-100 est additionné à la tension continue de 2,5 volts produite par le pont diviseur formé des résistances R1 et R2. Le zéro du signal est ainsi porté à 2,5 volts.

Un des fils alimentant l'appareil électrique (il peut s'agir pour cette expérience d'un appareil quelconque : lampe, appareil de chauffage, PC, réfrigérateur, etc.) traverse le capteur CT (ici : un TA12-100, mais tout capteur du même type fera l'affaire. Il faut cependant connaître le nombre de spires du bobinage). Attention : un seul fil doit traverser le capteur, non pas le câble complet).

Une résistance de charge est reliée aux deux bornes du capteur. Un pont diviseur avec deux résistances de même valeur (10 K ohms, la valeur a relativement peu d'importance, mais elle doit néanmoins être suffisante pour ne pas trop « tirer » sur l'Arduino) permet d'obtenir 2,5 volts à partir d'une broche 5 volts de l'Arduino. Cette tension continue, du fait du montage en série, s'ajoute à la tension aux bornes de la résistance de charge. Ainsi, la tension alternative est centrée autour de 2,5 volts, au lieu de 0 volt.

Le choix de la valeur de la résistance de charge va permettre de jouer sur la tension mesurée. Le but est d'obtenir une tension alternative comprise dans l'intervalle -2,5 à +2,5 volts. La loi d'Ohm intervient ici :

I = U / R

Ainsi, une résistance de 200 ohms permet de mesurer une intensité maximale de 2,5 V / 200 Ω, soit 0,0125 A.

I2/I1 = N1/N2 (voir plus haut)

N1 = 1, et N2 = 1000 (nombre de spires du TA12-100), donc I1 = I2 x 1000 = 12,5 A

Le courant est alternatif et, pour simplifier le calcul, supposé de forme sinusoïdale, donc l'intensité efficace est ici égale à 12,5 / RAC(2) = 8,84 A (voir l'explication de ce calcul dans le chapitre suivant).

Ceci correspond au courant circulant dans un appareil utilisant approximativement 2000 W (sous 230 V). C'est suffisant avec la plupart des éclairages ou des appareils courants. Si l'appareil fait plus de 2000 W, il faudra utiliser une résistance moins importante pour garder la tension mesurée en deçà de 2,5 V.

Le croquis est on ne peut plus simple :



Fig. 9 : Les données échantillonnées par le convertisseur ADC de l'Arduino, telles qu'elles apparaissent sur le traceur série.

```
void setup() {
  Serial.begin(2000000);
}

void loop() {
  Serial.print(analogRead(A0));
  Serial.println();
}
```

La boucle consiste à lire l'entrée analogique A0, puis à écrire la valeur lue (entier compris entre 0 et 1023) sur la sortie série. Le croquis est conçu avec l'efficacité en tête : il s'agit simplement d'échantillonner le courant le plus vite possible.

Compilez et téléversez le croquis, puis lancez le traceur série.

5.2 Mesurons le courant avec le capteur CT

Ici quelques notions sur le courant alternatif valent d'être rappelées. Pour faire simple, nous supposerons que le courant circulant dans l'appareil, dont nous souhaitons mesurer l'utilisation, varie selon une fonction sinusoïdale parfaitement en phase avec la tension. Ceci n'est totalement vrai que pour les appareils purement résistifs, tels que lampes, radiateurs, fers à repasser, chauffe-eaux ou bouilloires, etc. : le courant utilisé est égal à la tension divisée par leur résistance, point. En revanche dans le cas des appareils utilisant des composants tels que des moteurs ou des condensateurs, des phénomènes inductifs et/ou capacitifs entrent en jeu. Ces appareils renvoient de l'énergie dans le courant d'alimentation, d'où il résulte un déphasage entre la courbe du courant et celle de la tension. Des consommateurs plus complexes, tels que les alimentations de PC, ont une courbe de consommation de courant qui n'est plus sinusoïdale. Dans ce cas, si l'on souhaite obtenir une mesure précise, il faudra opter pour des techniques de mesure plus complexes, que nous n'aborderons pas ici faute de place.

Notons qu'ici intervient également la notion de facteur de puissance, qui permet de distinguer la puissance active (la puissance réellement produite par l'appareil, et facturée par le distributeur) mesurée en watts,

de la puissance apparente, égale au produit des valeurs efficaces du courant et de la tension (voir plus bas), et mesurée en VA (volts-ampères). Dans le cadre restreint qui nous intéresse ici de la mesure du courant consommé par un appareil purement résistif, ces deux valeurs sont égales. On a donc la puissance réellement consommée en calculant la puissance apparente. Dans tous les autres cas, la puissance active est inférieure à la puissance apparente et doit être déterminée par d'autres moyens.

Pour approfondir ces notions, nous vous recommandons la lecture de la page Wikipédia consacrée au facteur de puissance.

Dans le cas qui nous intéresse ici d'un courant alternatif sinusoïdal, l'intensité alterne selon une périodicité fixe entre une valeur négative minimale et une valeur positive maximale, les deux étant égales en valeur absolue. Nous notons I_{max} cette intensité, mesurée au sommet de l'onde sinusoïdale.

Dans le cadre de la mesure d'un courant alternatif, nous nous intéressons plus particulièrement au courant réellement consommé sur une période de temps. C'est ainsi qu'intervient la notion de courant efficace, ou courant RMS. Un appareil purement résistif consommant 1 A de courant efficace fournit la même énergie que s'il était traversé par un courant continu d'1 A.

Le courant efficace est défini comme la racine carrée de la moyenne de l'intensité au carré (RMS = *root mean square*), définie sur un intervalle de temps donné. En simplifiant, il s'agit de la moyenne du courant pris en valeur absolue. En particulier, lorsque le courant est de forme sinusoïdale, on démontre que l'intensité efficace est :

$$I_{rms} = I_{max} / RAC(2)$$

Concrètement, pour effectuer la mesure, nous allons utiliser un montage sensiblement différent par rapport à l'exemple précédent. Ici nous n'avons pas besoin de redresser le courant en déplaçant le 0 vers Vcc/2. En effet, il nous suffit de mesurer l'intensité maximale par rapport au 0. Nous n'utiliserons donc pas de diviseur de tension relié au +5 volts, mais une diode pour écrêter les alternances négatives. La diode sera placée en série avec le capteur, en amont de la résistance de charge de 200 ohms.

Le croquis utilisé pour l'expérimentation effectue une mesure du courant toutes les secondes. Le calcul suit les explications qui précèdent sur la mesure du courant alternatif. On notera en particulier la détermination de I_{max}, obtenue par échantillonnage en boucle de l'entrée A0 et mémorisation de la valeur maximale pendant un intervalle de temps égal à deux périodes.

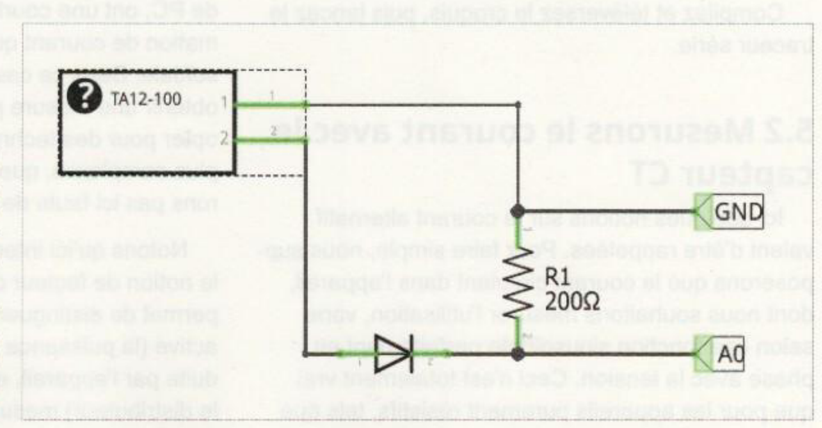


Fig. 10 : Le signal en sortie du TA12-100 est redressé au travers de la diode, pour ne conserver que les alternances positives. La résistance de charge R1 présente à ses bornes une tension proportionnelle au courant qui la traverse.

```
// Mesure de courant avec le capteur TA12-100

int capteur = A0;
// Le pic de tension aux bornes de la résistance
float tensionMax;
// Pic de courant au travers de la résistance
float courantMax;
// Courant efficace au travers de la résistance
float courantRMS;
// Valeur réelle du courant détecté par le capteur
float courantCapteur;

void setup()
{
  Serial.begin(115200);
  Serial.println("Go!");
  pinMode(capteur, INPUT);
}

void loop()
{
  tensionMax = getTensionMax();

  // Calcul du courant
  // Loi d'Ohm, résultat en mA
  courantMax = (tensionMax / 200.0) * 1000.0;
  // Division par RAC(2) pour obtenir le courant efficace
  courantRMS = courantMax / 1.4142;
  // Multiplier par le nombre de spires du capteur
  courantCapteur = courantRMS * 1000;

  Serial.print("Tension (pic) : ");
  Serial.print(tensionMax, 3);
  Serial.println(" volts");
  Serial.print("Courant au travers de la résistance (pic) : ");
  Serial.print(courantMax, 3);
  Serial.println(" mA");
  Serial.print("Courant au travers de la résistance (efficace) : ");
  Serial.print(courantRMS, 3);
  Serial.println(" mA");
  Serial.print("Courant détecté (efficace) : ");
  Serial.print(courantCapteur, 3);
  Serial.println(" mA");
  Serial.println();

  delay(1000);
}

float getTensionMax()
{
  int valeurLue;

  int valeurMax = 0;
  uint32_t start_time = millis();
  while((millis() - start_time) < 40) // 2/50 s {
    valeurLue = analogRead(capteur);
    if (valeurLue > valeurMax) {
      valeurMax = valeurLue;
    }
  }
  // Conversion de la valeur max en tension
  return (valeurMax * 5) / 1024.0;
}
```

Téléversons le croquis sur l'Arduino, puis affichons la console série. Si le courant circule dans le câble passant au travers du capteur (dans notre essai nous alimentons une ampoule halogène de 70 watts), nous devrions obtenir une sortie semblable à ceci :

```
Tension (pic) : 0.083 volts
Courant au travers de la résistance (pic) : 0.415 mA
Courant au travers de la résistance (efficace) : 0.293 mA
Courant détecté (efficace) : 293.480 mA
```

Le courant RMS apparaît sur la dernière ligne. En multipliant cette valeur par 230 volts (qui est la tension efficace du secteur en France, la tension maximale atteignant en valeur absolue 325 volts, c'est-à-dire 230 x RAC(2)), on trouve une puissance consommée de 67,5 watts, ce qui n'est pas éloigné des 70 watts inscrits sur l'ampoule.

5.3 Mesurons le courant avec un capteur à effet Hall

Reproduisons notre expérience précédente avec un capteur à effet Hall. Nous avons choisi d'utiliser plus spécifiquement un module à base d'ACS712, toutefois un autre modèle de capteur à effet Hall pourrait être utilisé.

Notre capteur, qui ressemble fort à un petit relais, possède d'un côté un bornier avec deux points permettant le montage en série avec le circuit électrique à mesurer. Contrairement au capteur CT, l'ACS712 nécessite donc une connexion directe avec le courant à mesurer. Bien sûr, nous vous recommandons comme toujours la plus grande prudence lors de son utilisation. Effectuez toutes les connexions hors tension, vérifiez deux fois avant de mettre sous tension, et ne laissez rien traîner sur la table. Travaillez dans un environnement tranquille, sans perturbation, mais non pas seul chez vous ! Si vous faites attention, vous ne prendrez pas plus de risque qu'en changeant une ampoule (et vous ne risquez pas de tomber de l'escabeau). Par ailleurs, ne dépassez pas le courant maximal spécifié par le fabricant, variable selon le modèle, et indiqué dans le *datasheet*. Vérifiez bien le marquage sur le boîtier (une bonne loupe vous sera peut-être nécessaire). On ne peut pas exclure qu'un vendeur peu scrupuleux vous ait fourgué un module équipé d'un ACS712ELCTR-05B-T (supportant un courant maximal de 5 ampères) comme étant un x20A ou un x30A. Une telle « erreur » pourrait entraîner des conséquences funestes (et « fumeuses »...).

Du côté opposé on trouve les trois broches détaillées plus haut dans le paragraphe introduisant les capteurs à effet Hall :

- Vcc doit être reliée à la broche 5 volts.
- GND doit être reliée à la broche GND.
- Vout doit être reliée à la broche A0.

Il n'y a pas de composant à ajouter, comme dans le cas du capteur CT : d'une part, le capteur génère en sortie une tension sur la broche Vout, qu'il suffit de mesurer directement (dans le cas du capteur CT une résistance de charge est nécessaire pour générer cette tension). D'autre part, par spécification, le signal généré sur cette même broche est centré sur Vcc/2, et borné entre 0 volts et Vcc, c'est-à-dire 5 volts. Il n'y a donc pas besoin de diode pour protéger l'entrée analogique.

Le croquis étant proche de celui utilisé dans l'expérience utilisant le capteur CT, nous ne mentionnerons que ce qui diffère (le croquis complet peut être récupéré sur le GitHub d'*Hackable*).



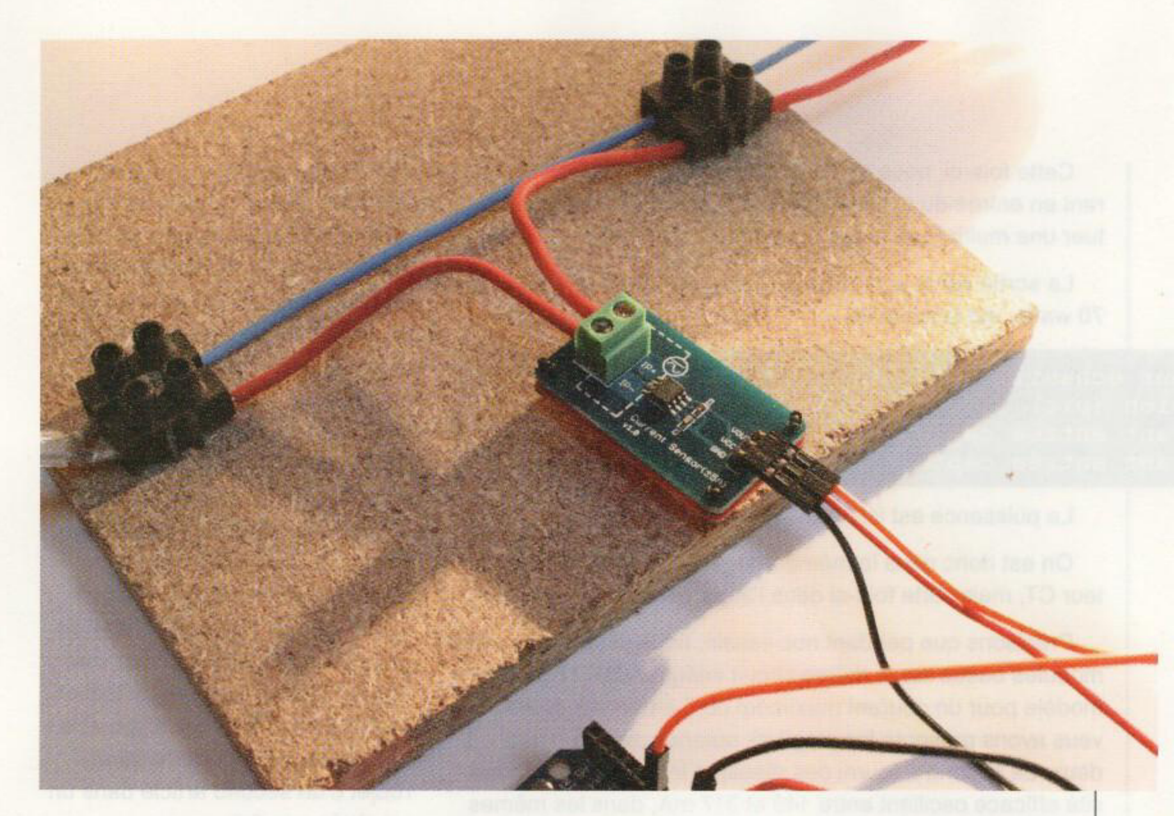


Fig. 11 : L'ACS712 en action. Comment ne pas rappeler d'effectuer toutes les connexions hors tension, avec les précautions d'usage ?

Tout d'abord, la valeur lue sur A0 étant centrée sur Vcc/2, nous avons besoin avant d'effectuer nos conversions, de la ramener dans un intervalle centré sur 0. La valeur à soustraire est théoriquement égale à 2,5 volts (Vcc/2), toutefois il est prudent d'expérimenter le montage en l'absence de courant à mesurer, afin d'effectuer un calibrage de ce point 0. Lors de notre expérience, en échantillonnant le capteur en l'absence de courant sur le bornier, nous avons obtenu une valeur de 516, soit légèrement supérieure à 1024/2. La valeur mesurée correspond donc à une tension au repos de 516 x 5 / 1024, soit 2,52 volts.

La dernière ligne de la fonction `getTensionMax()` devient donc :

```
return ((valeurMax - 516) * 5) / 1024.0;
```

Bien sûr, comme disent les anglo-saxons, « votre kilométrage peut varier ». Vous devrez donc effectuer votre propre calibrage, et remplacer 516 par la valeur que vous aurez obtenue.

La conversion de la tension en courant fait intervenir la sensibilité en mV/A fournie par la *datasheet*. Notre capteur est un ACS712ELCTR-05B-T, il est prévu pour un courant maximal de 5 ampères, et sa sensibilité est de 185 mV par ampère. Le courant en ampères est donc obtenu en divisant la différence de tension par rapport à Vcc/2 (que nous avons calculée précédemment), par le coefficient 0,185 :

```
courantMax = (tensionMax / 0.185) * 1000.0; // Résultat en mA
```

Le courant efficace est obtenu comme précédemment :

```
courantRMS = courantMax / 1.4142; // Division par RAC(2) pour obtenir le
courant efficace
```

Cette fois-ci, nous obtenons directement la valeur du courant en entrée du capteur (avec le capteur CT, il faut effectuer une multiplication par le nombre de spires).

La sortie sur la console série, avec la même ampoule de 70 watts, est la suivante :

```
Valeur échantillon = 533
Tension (pic) : 0.083 volts
Courant entrée capteur (pic) : 448.691 mA
Courant entrée capteur (efficace) : 317.275 mA
```

La puissance est ici : 0,317 A x 230 V = 72,91 W.

On est donc dans la même marge d'erreur qu'avec le capteur CT, mais cette fois-ci dans l'autre sens.

Précisons que pendant nos essais, nous avons testé deux modules basés sur le même circuit intégré ACS712 (le même modèle pour un courant maximum de 5 ampères). Nous vous avons présenté les résultats obtenus avec l'un des deux. Le second a fourni des résultats très instables (intensité efficace oscillant entre 149 et 317 mA, dans les mêmes conditions d'expérience). Étrangement ce dernier était parmi les moins chers...

POUR CONCLURE

Précisons encore que tous ces essais ont été faits uniquement avec un courant traversant une charge résistive (une ampoule de 70 watts), donc un courant de forme sinusoïdale, en phase avec la tension. Dans ces conditions, les résultats obtenus sont plutôt satisfaisants. Un calibrage plus fin du montage, tenant compte des caractéristiques mesurées de l'entrée analogique de l'Arduino, de la résistance et de la diode dans le cas du capteur CT (les diodes à l'état passant présentent une chute de tension qui constitue une de leurs caractéristiques), permettrait sans aucun doute d'obtenir une mesure plus précise.

Toutefois, à l'échelle du domicile, notre utilisation de l'énergie électrique ne se limite plus depuis très longtemps aux ampoules à filaments, aux chauffages à résistance et autres fers à repasser ou chauffe-eaux. La présence d'équipements tels que réfrigérateurs, PC, télévisions, climatisations, VMC, machines à laver, etc. fait que le courant circulant dans les circuits n'est plus en phase avec la tension, et que la puissance en résultant s'écarte du modèle d'onde sinusoïdale dont il est question dans cet article. Les fonctions mathématiques de calcul de courant ou de tension alternative n'ont plus tout à fait cours et donnent un résultat approximatif. Pour obtenir une mesure vraiment précise, il faut faire appel à des algorithmes d'échantillonnage un peu plus complexes, permettant de mesurer la puissance active réellement consommée dans toutes les situations, en mesurant en même temps le courant et la tension. Il nous a semblé plus pédagogique d'appréhender le sujet par étape, et de vous présenter la façon de faire la plus simple possible. Une mise en application plus orientée vers des conditions réelles d'utilisation, exploitant les librairies disponibles dans le domaine, pourra faire l'objet d'un second article dans un prochain numéro.

Notons que si vous avez besoin simplement de détecter la mise en fonctionnement d'un appareil (sans s'intéresser à sa consommation), quel qu'il soit (par exemple : mise en fonctionnement d'une pompe de forage ou de piscine, d'un PC...), nul besoin d'une précision « au 1/10ème » : les montages ci-dessus conviennent parfaitement.

Cet article nous a permis toutefois d'ouvrir les portes de la détection et de la mesure du courant électrique et du champ électromagnétique, et d'entrevoir les nombreuses applications pouvant exploiter ces phénomènes, et pas seulement dans le cadre de la mesure du courant. Les capteurs à effet Hall ouvrent notamment des perspectives d'emploi que l'on imagine aisément assez vastes, en matière de contrôle, de sécurité, de détection ou de mesure. **ELB**



VOYAGE AU CENTRE DU VIRTUEL

Nantes • Du 14 au 17 novembre 2017

La Cité,

le Centre des Congrès de Nantes

www.jres.org | #JRES2017

Design Graphique : Nicolas gazut | www.iamcion.com